VIC
REVEALED

NICK HAMPSHIRE

VIC™
REVEALED

NICK HAMPSHIRE

DUCKWORTH

INTRODUCTION

This book is a collection of discoveries about the VIC, how and why it works, and how to use these facts to write better programs and perform more interesting functions. The book is divided into five sections, each section covering one of the principal functional blocks into which the basic VIC computer can be divided. The different aspects of the VIC dealt with in each section cover most of the advanced applications for which the VIC can be used.

The VIC is produced in several slightly different versions, in different parts of the world. This book is written for version 7 machines which are designed for use with European PAL TV sets. The US version 6 machines use the 6560 VIC chip which is compatible with the US 525 line TV. The Japanese version 1 machines use the 6560 and also have Japanese character keyboard and character generator. There are slight differences in the operating system software of these three versions but they do not affect most of the information in this book.

I should like to thank Commodore UK and Commodore US for their assistance in writing this book, in particular the following people: John Baxter and Malcolm North of CBM UK and Mike Tomczyk, Shiraz Shivji and Bob Russell of CBM US.

Nick Hampshire
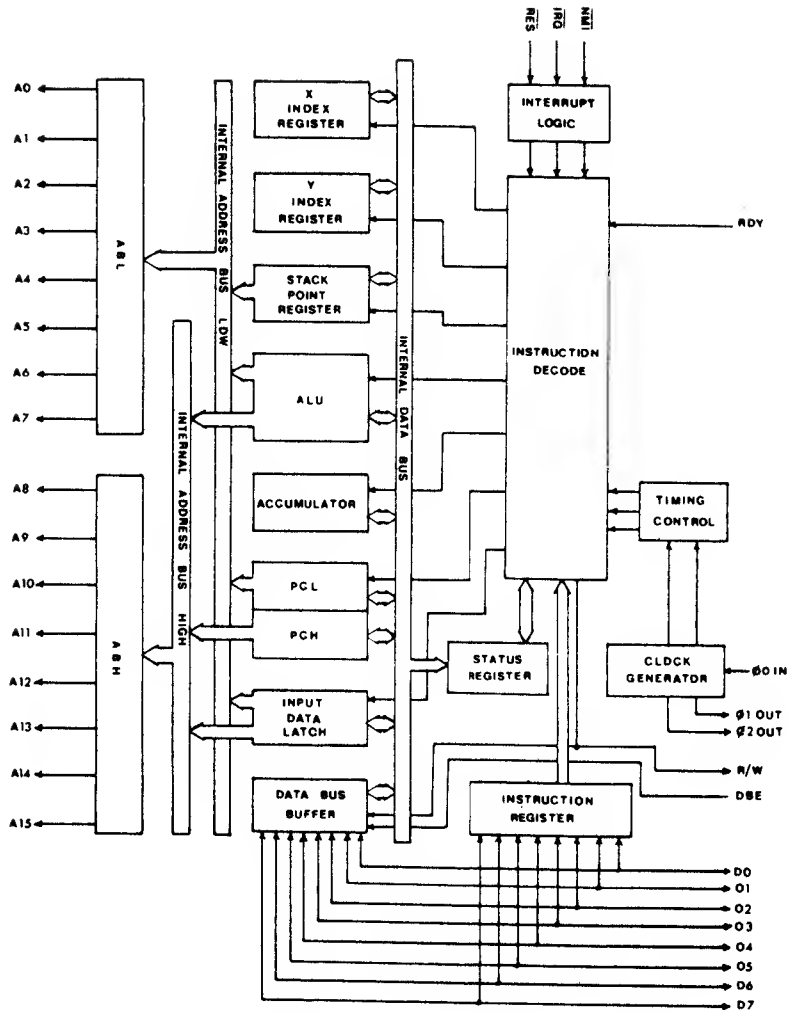
CONTENTS

1 THE 6502 MICROPROCESSOR

Fig. 1 — Block diagram of internal structure of 6502.

1

THE 6502 MICROPROCESSOR

When a program is run on the VIC all the instructions are performed by one component, the microprocessor. There are a range of different microprocessors, this particular device is manufactured by MOS Technology and known as the 6502. It is an eight bit microprocessor, eight bits meaning that during each instruction or operation cycle, eight bits of data are operated upon or transferred simultaneously.

A block diagram of the internal structure (known as the system architecture) is shown in Figure 1. This may appear rather complex, but it can be divided into two sections. One called the control section, the other the register section. The control section lies on the right side of the drawing, the register section on the left. All the processing is carried out within the register section of the chip, instructions obtained from program memory are implemented by a series of data transfers within this section. Each of the 56 different instructions which the 6502 recognises involves a unique set of data transfers. It is the control section which recognises the instruction, and initiates the correct sequence of data transfers. The instructions enter the processor via the data bus and are latched into the instruction register to be decoded by the control logic. Since most instructions require more than one data transfer within the register section, a source of timing signals is required to ensure the correct sequence, this is done by the timing control unit.

Each data transfer which takes place within the register section, is the result of the decoding of the instruction register and the timing control unit by the control logic, whose outputs enable the relevant registers. When programming at a machine level a primary concern is the control and manipulation of data within the processors registers. To understand the function of the microprocessors instruction set, one must understand the function of its registers.

MEMORY USAGE AND THE 6502 INSTRUCTION CYCLE

The 6502 microprocessor has a 16 line address bus, this enables it to access 2^{16} or 64K of memory. Any one of the 65,536 memory locations can be accessed by the processor placing the correct binary value corresponding to the memory location on the address bus. The eight bits or byte of data located at the addressed memory location can then be read, or if required changed, via the eight line data bus. Since all the processor registers and memory are only eight bits long it requires two bytes to specify a sixteen bit memory address. The bottom eight bits are referred to as the high order address. By dividing the 16 bit address into two 8 bit sections the entire addressable memory area can be split into logical blocks, or pages. Memory within each page can be addressed using the low order address byte, each page has 256 memory locations and there are 256 pages. Page zero starts at location 0 and ends at address 255, page one goes from address 256 to 511. Apart from two important exceptions the concept of paging is not important to the programmer, these being, page one which contains the processor stack, and page zero which has special addressing modes.

It is usual to express memory addresses and their data contents in hexadecimal notation, this being easier to write than binary, yet more easily converted into binary than a decimal value. The convention is to identify hexadecimal values by preceding the value with a dollar sign, this prevents any confusion as to whether the value is in hexadecimal or decimal. In hexadecimal any address is represented as a four digit value, the first two digits being the high order address byte and the bottom two digits the low order address byte. The paging concept is thus clearly seen in a hexadecimal address. Any data value is represented in hexadecimal as a two digit value.

Memory is used by the processor for the storage of both programs and data, the data can be either included within the program, usually as constants, or in separate data tables. Programs can be stored either in RAM or ROM memory but variable data can only be stored in RAM memory. Each instruction in a machine code program requires between one and three bytes of memory. With a one byte instruction the data on which it operates is stored in one of the processor registers. A two byte instruction consists of the instruction first followed by a one byte operand, this can be either a zero page address or a data constant. An instruction occupying three bytes contains the

3
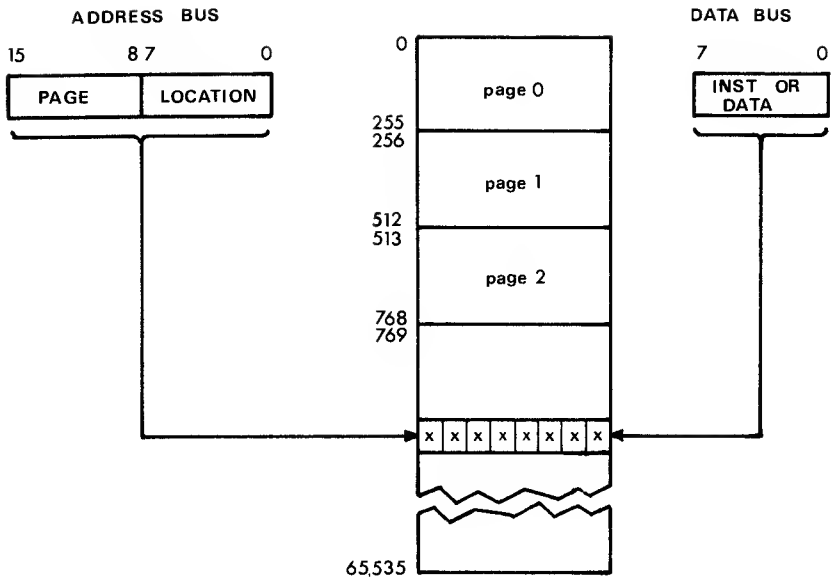
Fig. 2 — Relation between address, data, and memory location.

Instruction	Memory Contents	Function	Start Address
LDA#$02	$A9	INST	+ 1
	$02	OP	+ 2
STA $0253	$8D	INST	+ 3
	$53	OP LSB	+ 4
	$02	BP MSB	+ 5
CLC	$18	INST	+ 6
ADC #$50	$69	INST	+ 7
	$50	OP	+ 8
STA z$55	$85	INST	+ 9
	$55	OP	+ 10
RTS	$60	INST	+ 11

Fig. 3 — How a program is stored in memory.

4

instuction followed by a full sixteen bit address in the form; low order byte followed by high order byte.

When the processor executes the program instructions stored in memory it goes through a fixed instruction cycle, this gets the instruction from memory, performs the instruction, and then repeats by getting the next instruction, and so on until the end of the program. There are three stages in the instruction cycle, they are;
1) fetch the instruction.
2) decode the instruction.
3) execute the instruction.
Fundamental to the operation of the instruction cycle is the internal processor register called the program counter. The program counter holds the 16 bit address of the next instruction, and the first stage in the instruction cycle is to transfer the contents on the program counter onto the address bus. The instruction located at that memory address is then transferred to the processor instruction register. The second phase of the cycle is to decode the contents of the instruction register to generate the correct sequence of internal and external signals to perform the execution stage of the cycle. The execution phase of the cycle depends on the instruction and will include the fetching of any operand bytes plus the manipulation of one or more processor registers. After fetching an instruction or an operand byte the program counter is incremented by one so that at the end of the instruction cycle it contains the address of the next instruction and the process is repeated.

Fig. 4 — Sequence of processor operations in executing an instruction.
  Step 1 — program counter points to location of instruction by placing the memory address on the address bus.
  Step 2 — the instruction code is transferred from memory to the instruction register where it is decoded.
  Step 3 — the program counter is incremented to point to the operand byte of the instruction in the following memory location, this byte is placed in the accumulator. The decoded instruction then results in a specific operation being performed on the byte in the accumulator.
  Step 4 — The program counter is incremented to point to the next instruction in memory and the sequence returns to step 1.
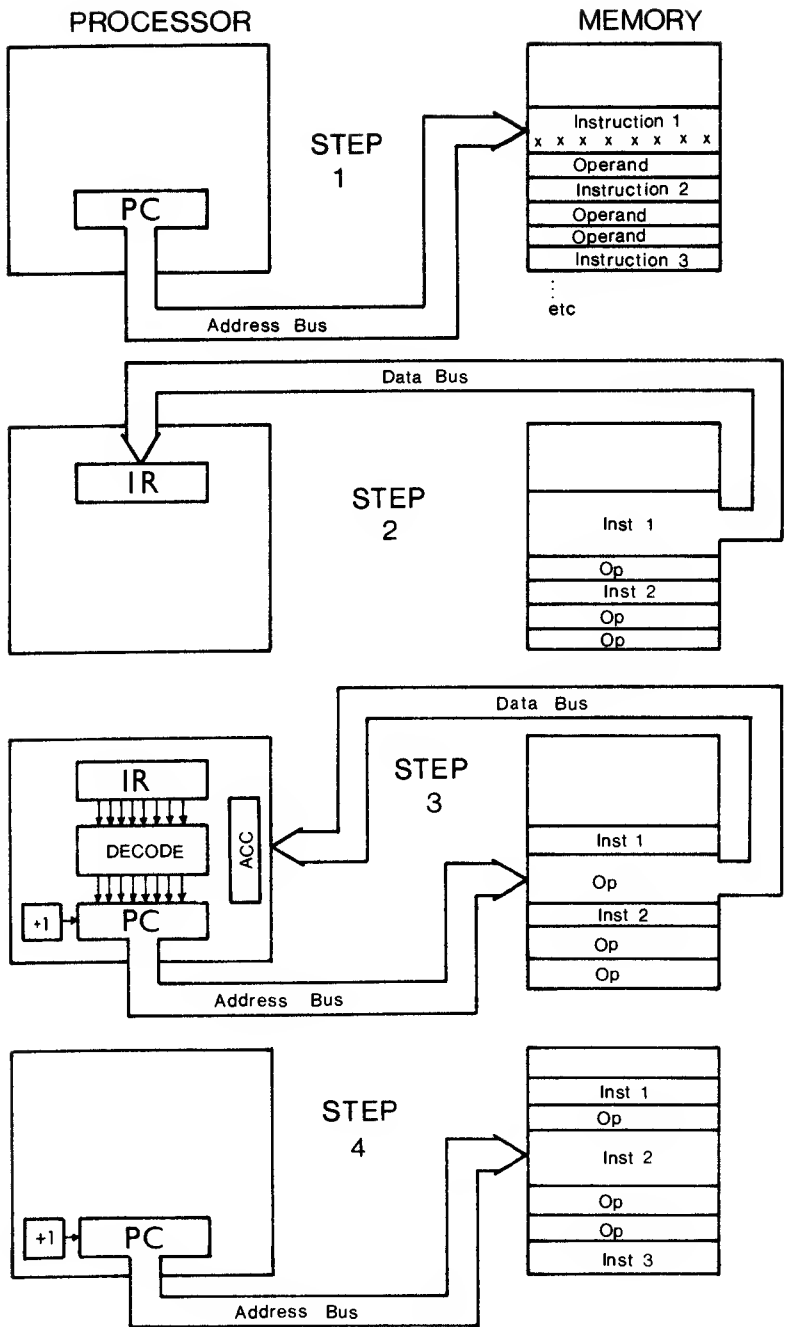
Fig. 4

THE ACCUMULATOR AND THE ARITHMETIC UNIT

The accumulator has no exact function, it is a general purpose register. To move a byte of data from one part of memory to another it must be temporarily stored in the accumulator. The accumulator is also used to store the intermediate and final results of a logic or arithmetical operation.

Data transfers between the accumulator and memory (since the VIC is a memory mapped system this also includes I/O) are very important and account for about 40% of all the instruction used in a machine code program. To move a byte of data from one memory location to another requires two instructions:

LDA,M1 — Load accumulator with contents of first memory location.

STA,M2 — Store contents of accumulator in second memory location.

Memory locations M1 and M2 are accessed by one of a variety of addressing modes, these will be looked at later in this section. Having loaded a byte of data into the accumulator the processor can be instructed to perform arithmetic or logical operations upon it. Only about three percent of all instructions in a program are arithmetic or logical operations.

Since the 6502 is an eight bit machine all the arithmetic and logical operations are between two eight bit numbers. The arithmetic or logical operation is performed in the ALU or arithmetic logic unit, this requires that one of the operands is in the accumulator and the other is in a memory location. The result of the operation is placed in the accumulator. Placing the results in an eight bit accumulator causes a problem when adding two numbers whose sum is greater than 255. This is overcome by giving the accumulator a ninth bit, called the carry. The carry bit, or flag, is one bit in the processor status register, and is set when the contents of the accumulator exceed 255. This applies to the performance of binary arithmetic by the processor, the 6502 is fairly unique in that it can also do decimal arithmetic. In this mode each byte contains two binary coded decimal numbers, numbers from 0 to 99 can be stored as a single byte. As in the binary mode, the carry flag is set when the addition of two numbers gives a result greater than 99. The processor is placed in the decimal mode by a "set decimal mode" instruction, SED, which sets another bit within the processor status register.

There are two basic arithmetic instructions, ADC — which is "add memory to accumulator with carry", and SBC — which is "subtract memory from accumulator with borrow". Both instructions can be either binary or decimal in nature and can use a variety of addressing modes to indicate the memory location.

The ADC instruction adds the value of the data in the memory location, plus the carry from the previous operation, to the value in the accumulator, storing the results in the accumulator. If the result exceeds 255 in the binary mode, or 99 in the decimal mode, then the carry flag is set, if the result is zero then the zero flag is set. An example, if we want to add the two numbers, 25 and 189, and store the result in memory location 10 (decimal) we could use the following sequence of instructions:

CLC	18	(this clears the carry flag)
LDA 25	A9 19	(Load accumulator with 25)
ADC 189	69 BD	(Add 189 to accumulator and carry)
STA 10	8D 0A 00	(Store result in location 10)

The instructions in the left column are in mnemonic code, followed by a decimal number or memory location. The same sequence of instructions appears on the right as hexadecimal values. Addition of two numbers where one or both have values greater than 255 needs a process known as multiple precision addition, calling for the use of the carry flag. Adding two sixteen bit numbers, requires two additions. The carry is first cleared and the two lowest order bytes, (a sixteen bit number would be stored in two bytes of memory) added together. The result of this addition is stored in a memory location as the low order byte of the result. Now the two high order bytes are added, plus any carry generated by the first addition, the sum stored as the high order byte of the result. Using this method numbers of any size can be added together, whether the processor is in binary or decimal mode.

Addition can be performed on signed numbers. Positive numbers added to negative numbers, or two negative numbers added. The sign is stored as bit seven of the highest order byte, a zero for positive and a one for negative. Addition takes place as in ordinary arithmetic, the only exception being that the carry flag for the highest order byte is replaced by the overflow flag. This performs the same function but records an overflow or carry from bit seven, rather than bit eight. Negative numbers are stored not as ordinary binary numbers but as

two's complement, which is best described as the inverse of that number minus one. All the ones become zeros and vice versa for all bits, except bit one, thus binary five is normally 00000101 — in two's complement form it become 11111011.

The SBC instruction subtracts the value of data in a memory location (and borrow) from the value in the accumulator, storing the result in the accumulator. Two's complement arithmetic is used throughout. The borrow flag is the same as the carry flag used in addition, whereas before an addition the carry flag is always cleared, before a subtraction it is always set. The result of subtraction affects the carry or borrow flag, it is set if the result is greater than or equal to zero. Similarly for subtraction of signed numbers the overflow flag is set if the result exceeds +127 or —127 for single precision seven bit arithmetic. The SBC instruction can be used with either binary or decimal numbers with both multiple precision and signed arithmetic. To subtract two decimal numbers, say, 18 from 27 use the following sequence of instructions, the decimal mode is used to illustrate its function:

SED	F8	(set decimal mode instruction)
SEC	38	(set borrow flag)
LDA 27	A9 27	(load accumulator with 27)
SBC 18	E9 18	(subtract 18 from accumulator and borrow)
STA 10	8D 0A 00	(store result in location 10)

The instructions on the left are in mnemonic code, on the right in hexadecimal, note that in the decimal mode the hexadecimal and decimal numbers are the same.

The 6502 instruction set does not include instructions to perform multiplication or division. Users requiring these functions must write special subroutines to perform them, or use the subroutines within VIC Basic. Multiplication is a process of repeated addition: 3 x 5 is the same as 5 + 5 + 5. For large numbers this could be a lengthy process, and programming tricks are required to minimise this. Division is a process of repeated subtractions: 15/5 can be performed as the following sequence, 15 — 5 = 10, 10 — 5 = 5, 5 — 5 = 0, since three subtractions were required, the answer is 3. As with multiplication, programming techniques are needed to reduce the time taken to divide large numbers.

Besides arithmetic operations the ALU can perform logical operations between data in memory, and the accumulator. There are

four such instructions AND, BIT, OR and EOR. The AND instruction performs a bit by bit logical AND operation between a memory location and the accumulator, storing the result in the accumulator. This operation can be used to reset or mask a single bit or group of bits in a memory location. In the decimal mode each byte holds two digits, the AND instruction can be used to extract one digit. Where there is a zero in the operand, there is a zero in the result. To mask out the most significant decimal digit stored in the bottom four bits, the accumulator is ANDed with 00001111 or hexadecimal OF

LDA 25   A9 25   (load the accumulator with decimal 25)
AND 0F (hex) 29 F0   (AND the accumulator with 00001111 binary)
STA 10   8D 0A 00 (store the result in location 10)

On running this program location 10 will contain 05, the 2 being masked out and replaced by a 0. The BIT instruction is identical to the AND, except that the result is not stored in the accumulator and only the status register flags are set.

The OR instruction performs a binary OR on a bit by bit basis between the contents of the accumulator and a memory location, the result is stored in the accumulator. The main use of this instruction is to set a bit or group of bits in a memory location, a logical 1 in the operand field produces a 1 in the corresponding bit of the result. The EOR or "Exclusive OR instruction" is identical to the OR, except that a logical 1 appears in the result only if there is a 1 in the operand field, and a 0 in the accumulator for the corresponding bit. The main use of the EOR instruction is to produce the two complement of a byte.

ADDRESSING MODES

Every instruction in a machine code program contains information on the position of the data on which that instruction will operate. The same instruction can exist in several forms depending on where the data is located and each of these forms is referred to as an addressing mode. There are thirteen different addressing modes and most instructions can be performed in more than one mode. The LDA instruction can use one of eight different modes of addressing. The thirteen address modes can be divided into seven basic modes and six modes which are combinations of one of the basic modes and the indexed addressing mode.

Implied
Accumulator
Immediate
Absolute
Zero page
Relative
Indirect

Absolute X indexed
Absolute Y indexed
Zero page X indexed
Zero page Y indexed
Indirect indexed
Indexed indirect

The simplest mode is Implied addressing, which is used exclusively by single byte instructions operating on the internal processor registers. In an instruction like CLC (Clear Carry) no data is accessed therefore no address is required. It is implied that a register, in this case the Status Register, is to be operated upon.

Accumulator mode addressing is used in instructions which perform logical operations on data in the accumulator. This mode is a version of implied addressing and all instructions are single byte.

Immediate addressing is used whenever the programmer wants to perform an operation using a constant. To put a value of, say 25, in the accumulator we would use the LDA instruction in the Immediate mode. In this form of addressing the data is stored in the byte immediately following the OPCODE.

11

Neither the Immediate or Implied addressing modes use a memory address where data is stored, and are of little use in operations with variables. To address any location in memory would require a full sixteen bit or two byte address stored in the Operand part of the instruction. This address points to a memory location where the variable upon which the operation being performed is currently located, or is to be stored. This form of addressing is known as "Absolute addressing".

A shortened form of absolute addressing can be used when the memory location being accessed lies on page zero of memory. This is the only case where the concept of paging has any importance in the 6502, page zero is just the bottom 256 memory locations. This is called "Zero Page Addressing", and uses a single byte address to point to the location of data within page zero. A two byte Zero page address mode instruction is much faster than three byte Absolute addressing and it is good practice to store all variables in page zero. When running machine code programs on the VIC only the bottom 144 bytes of page zero should be used, storing data in locations above this will probably cause the machine to crash.

A special form of addressing is used exclusively by branch and jump instructions, known as "Relative addressing". In this addressing mode the instruction is followed by a single byte Operand. This does not specify an address as in zero page addressing, but a displacement from the address where the branch instruction is stored. Since the displacement must be either positive or negative, bit eight is used to signify the jump direction, this allows the jump to be up to 127 bytes forward or 128 bytes backward.

In some programs it may be necessary to have a computed address rather than a fixed address, as in absolute addressing. This is done using indirect addressing, instructions in this mode have just a single eight bit address field which points to the effective address as two bytes in page zero. The data address is thus not stored directly in the Operand field of the instruction, but, indirectly in page zero, all the indirect addresses are indexed except for the JMP instuction.

Indexed addressing uses the contents of one of the two index registers as an offset to the address stored as the Operand part of the instruction. The address stored in the Operand can be either an absolute two byte address, or a zero page single byte address. This

gives a total of four different indexed addressing modes, two for each index register. The primary use of indexed addressing is in the access of successive memory locations used for the storage of a table or block of data.

THE PROCESSOR STATUS REGISTER AND THE USE OF FLAGS

The processor status register occupies a very important position in the system architecture of the 6502. It is an eight bit programmable register, unlike the other registers and its function lies between the control and register section of the processor. It is the only register which actually affects the control logic. Seven of the eight bits are used, and each bit, or flag, has a specific function.

Flags fall into three categories, those controllable only by the programmer, those controllable by both programmer and processor, and lastly those controlled solely by the processor. Only one flag falls into the first catagory, the Decimal mode or D flag, occupying bit three of the status register. This flag controls whether the processor performs binary or decimal arithmetic. It can be set by a SED instruction, after which all arithmetic is performed in the decimal mode, until the D flag is cleared by a CLD or clear decimal mode instruction.

Three flags fall into the second category: Carry, Overflow and Interrupt disable. The Carry or C flag is located in bit 0 of the status register, it is modified either by the results of certain arithmetic operations or by the programmer. The carry is also used as a ninth bit during arithmetic operations or by the shift and rotate instructions. The instruction used to set the carry flag is SEC, it can be cleared by CLC.

The Overflow or V flag occupies bit six of the status register, and is used during signed binary arithmetic to indicate that the result was of greater value than could be contained within the seven bits of the signed byte. The V flag has the same meaning as the carry flag, but also indicates that a sign correction routine must be used if this bit is "on", since the overflow will have erased the sign in bit seven. The programmer can only clear the V flag by using the CLV instruction.

The Interrupt disable, I flag, controls the operation of the microprocessor interrupt request input, and is located in bit two of the status register. Interrupts play a very important part in the VIC's design, and each time there is an interrupt the I flag is set by the processor. This stops the processor being interrupted by more pulses on the IRQ line until the interrupt handling program has been completed with a return note an interrupt instruction clearing the I flag.

14

The I flag can also be set by the programmer with an SEI instruction to prevent the processor being interrupted, as during a precision timed loop subroutine. At the end of such a program the interrupt line can be returned to its normal function by clearing the I flag with a CLI instruction.

The last three flags: Zero, Negative and Break, are controlled solely by the processor. The Zero and Negative flags are either set or reset by nearly every processor operation. The Zero or Z flag is set by the processor whenever the result of an operation is 0, as when two numbers of the same value are subtracted from each other. The Negative or N flag is set equal by the processor to bit seven of the result of an operation. One of its primary uses is during signed binary arithmetic, if the N flag is set then the result is a negative number. The Break or B flag is set by the processor during an interrupt service sequence. The Z flag occupies bit one, the N flag bit seven and the B flag bit four of the status register.

The seven status bits or flags in the status register each have a meaning to the programmer at a particular point in the program. Although the carry and overflow flags are used in arithmetic operations the major use of flags is in combination with the conditional branch instructions. This gives the programmer the capability of incorporating decision making instructions within a program, to test a flag, and, depending on the state of that flag, take one or two courses of action. A conditional branch is functionally the same as the IF... THEN GOTO... statement in Basic. There are a range of these instructions performing different functions and testing different flags. Anyone writing a machine code program must keep track of the expected state of all flags at every instant throughout the program. Failure to do this is one of the commonest causes of a program not working or producing the wrong result. An example would be failure to clear the carry flag before an addition. On odd occasions it would have been set by a previous instruction, and thus give rise to erroneous results.

BRANCHES, JUMPS AND THE PROGRAM COUNTER

To understand the use of branch and jump instructions the concept of program sequencing must be understood, and its control by the program counter. The program counter, or PC, consists of two eight bit registers. Like the other registers they communicate with the internal processor data bus, but the outputs are also connected to the sixteen lines of the address bus. One of the PC registers is connected to the bottom eight address lines and is called PCL, the other, the PCH is connected to the eight high order address lines. Although two eight bit registers, they function like a single sixteen bit register. It is the program counter which controls the addressing of memory, by being a program, or data address pointer and, as such it contains the address of the next memory location to be accessed.

At the beginning of the program the PC must contain the address of the first instruction. This is one of the functions of the operating system reset software. It is also preformed by the SYS and USR commands when entering a machine code program from Basic. The instruction fetched from memory is stored in the instruction register, to be decoded by the control logic. This process takes one clock cycle, during which time the program counter is incremented by one to point to the next memory location. The processor usually requires more than one byte to interpret an instruction, this first byte contains the basic operation and is known as the OP CODE. The following one or two bytes, known as the OPERAND, contain either a byte of data or the address of the data on which the operation will occur. An instruction may require up to three sequential memory locations, the program counter first points to the OP CODE which is fetched from memory and stored in the instruction register. The PC is incremented and points to the next memory location, the contents of which are fetched and stored in the ALU, in a three byte instruction this will be the low order address of the data. The program counter is again incremented and the high order address fetched from the third memory location. The processor then latches the two bytes of the address onto the address bus via the ALU, fetches the data, and performs the operation. Having completed the operation, which usually takes about four clock cycles, the processor increments the program counter to point to the next instruction and the process is repeated. In this manner the program counter will continue to advance until it reaches the maximum memory location, fetching instructions and addresses.

A sequential program would lack a feature fundamental to computing, the ability to test the result of an operation, and implement various options based on the results of the test. Firstly flags can be used to test the result of an operation, secondly the contents of the program counter must be changed to point to the start of a new program. The simplest way of changing the contents of the program counter is with the JMP or "Jump to new location" instruction. This as its name implies does not perform any tests on the results of a previous operation. It simply loads a new sixteen bit address into the program counter thereby forcing the processor to start operating at the new address. The JSR or "Jump to Subroutine" instruction is similar to JMP except that the current contents of the program counter are saved on the stack to be restored on the completion of the subroutine by an RTS, "Return from Subroutine" instruction.

There are eight different conditional branch instructions, they can be divided into four groups, each testing the state of one of the status register flags. The four flags tested by the conditional branch instructions are: Carry, Zero, Negative and Overflow, one instruction tests if the flag is set, and the other if it is clear. The two instructions for the Carry flag are BCC or "Branch on Carry Clear" and BCS or "Branch on Carry Set". The Operand contains the address to which the program jumps if the condition being tested is true. The addressing mode used is unique to conditional branch instructions, it is called relative addressing.

In relative addressing the new address is stored as just one byte, which is added to the current contents of the program counter. To enable the program to branch both forwards and backwards the relative address can be either a positive or a negative number. The fact that relative branch addresses are stored as a signed single byte limits the maximum size of the branch to either 127 bytes forwards or 128 bytes backwards, this may seem a limitation but in practice it is not.

The eight conditional branch instructions are:

BMI — Branch on Result Minus

                                                Testing the N flag

BPL — Branch on Result Plus

BCC — Branch on Carry Clear

                                                Testing the C flag

BCS — Branch on Carry Set

BEQ — Branch on Result Zero

$\left.\right\}$ Testing the Z flag

BNE — Branch on Result Not Zero
BVS — Branch on Overflow Set

$\left.\right\}$ Testing the V flag

BVC — Branch on Overflow Clear

Most operations involve the setting of one or more flags, but a small group of test instructions are specifically designed to set flags for testing by a branch instruction. The most commonly used is the "Compare Memory and Accumulator" or CMP instruction. It allows the programmer to compare a value in memory to one in the accumulator without altering the value in the accumulator. If the two values are equal the Z flag is set, otherwise it is reset. The N flag is set equal to bit 7 and the carry flag is set when the value in memory is less than or equal to that in the accumulator. The BIT instruction tests single bits in memory with the corresponding bits in the accumulator.

THE STACK REGISTER AND ITS USE

The stack register is mainly concerned with the handling of interrupts and subroutines. It is an eight bit register, its function is identical to that of the program counter since it is an address generator. It is used to point to an address in page 1 of memory, (locations 256 to 511), known as the "Stack". The stack is a set of memory locations starting at 511 and filled downwards from that location with a maximum size of 256 bytes. It is organised as a LIFO or "Last In First Out" structure, which means that the last byte of data stored on the stack is the first byte to be accessed. Every time data is pushed onto the stack the stack pointer is decremented by one, and each time data is pulled off the stack, the stack pointer is incremented by one. The addressing of the stack is independent of the program and based purely upon chronological events. The stack is used as a temporary data store, the most common data being re-entrant addresses generated by subroutines and interrupts.

Every time a subroutine is called in a machine code program the current contents of the program counter are saved. On returning from the subroutine the program can be re-entered at the correct location. Similarly every time the processor is interrupted the current address in the program counter is saved before the processor performs the interrupt servicing routine. A subroutine may call other subroutines, requiring the storage of several re-entrant addresses in the stack. The last re-entrant address stored is the first address reloaded into the program counter at the end of the subroutine, hence the LIFO structure of the stack. The calling of subroutines by other subroutines is termed "subroutine nesting" and is a common occurrence in machine code programs. The size of the stack in the 6502 limits the user to 127 levels of nesting, usually far more than is needed. Basic subroutines also use the stack for the storage of the return address pointers and register contents.

A subroutine is called by a JSR or "Jump to Subroutine" instruction. This pushes the current contents of the program counter onto the stack. A location stored as the Operand field is then loaded into the program counter. This causes the processor to jump to a new section of the program and start execution from the location in the program counter.

The return from a subroutine to the main program is accomplished by he RTS or "Return from Subroutine" instruction. This loads the

19

return address from the stack into the program counter. It also increments the program counter to point to the instruction following the JSR. The stack pointer is also incremented to point to the next subroutine address if any.

The stack can be used by the programmer as a temporary storage location for data passed to a subroutine. The programmer needs a set of instructions to allow him to put data onto the stack and read it back. The current contents of the accumulator can be transferred to the next location on the stack by the PHA or "Push Accumulator onto Stack" instruction. Data can be read from the current location pointed to by the stack pointer into the accumulator, by the PLA or "Pull Accumulator from Stack" instruction. Both instructions automatically cause the stack pointer to be incremented or decremented by one. An example of data storage in the stack is saving the contents of the processor status and index registers when a subroutine is called. The contents of the status register can be pushed onto the stack by the PHP "Push Processor Status on Stack" instruction and then transferred from the stack back to the status register by the PLP "Pull Processor Status from Stack" instruction. To save the contents of the index registers they are first transfered to the accumulator and then placed on the stack. When writing any machine code routine for the VIC which will be called from a Basic program it is very important to first save the contents of the processor accumulator and index registers on the stack. The contents of these registers are then restored prior to returning to Basic. Failure to do this will result occasionally in system crashes.

Normally the stack pointer points to a location in page one, the location being automatically incremented or decremented by the processor as required, but in some situations the programmer has to be able to change the stack pointers contents. The stack pointer is loaded by transferring the contents of the X index register to the stack pointer with a TXS "Transfer Index X to Stack Pointer" instruction. This instruction is used at the beginning of a program to initialise the stack pointer, it is performed automatically on the VIC as part of the power up reset routine. Re-initialising the stack on the VIC causes problems, usually resulting in a crash and should thus be avoided. The current contents of the stack pointer can be read by loading it into the X index register with a TSX "Transfer Stack Pointer to Index X" instruction.

THE INDEX REGISTERS

Having a fixed address in the Operand field of an instruction poses problems when accessing a sequential block of data, such as a table or an input buffer. One method would be to use a string of load instructions in the form, load data from address 1 — perform operation — load data from address 2 — perform operation and so on. This is obviously highly wasteful of memory space, it would be more efficient if this program was written as a loop. To do so would require that the address stored as the Operand field of the load instruction is incremented each time the program goes round the loop. In this way the Operand address will always be pointing to the next byte of data to be accessed. This method is useful, but, execution time is considerably greater than in the straight line programming technique, also it is often undesirable to use a self modifying program.

A more sophisticated approach is the use of a counter, the contents of which are automatically added to the address in the Operand field of the instruction. Such a counter is called an Index register. There are two Index registers in the 6502, both are eight bit registers, labelled X and Y. They are used by instructions in one of the indexed addressing modes. The simplest is absolute indexed addressing, in this mode the contents of one Index register is added to the address in the Operand field of the instruction, giving a new address from which data is to be accessed. The fact that the Index registers are only eight bit registers limits the maximum size of data block accessed using indexed addressing to 256 bytes. In practice the majority of tables are shorter, it is not therefore a significant limitation. If longer tables are required then programming techniques, such as indirect indexed addressing, are used to overcome this limitation.

The Index registers are controlled and manipulated by a range of special instructions. A number can be loaded to, or stored from the Index register and a memory location, by the LDX, LDY and STX, STY instructions. Similarly the contents of the Index registers can be compared with a value in memory to test if a conditional branch should take place by using the CPX and CPY instructions. The contents of an Index register is changed to point to the next address by increments or decrementing it by one. To count up, the instruction used is INX or INY, to count down DEX or DEY. The remaining Index register instructions allow the transfer of the contents of the accumulator into one of the Index registers and vice versa. TAX and

21

TAY transfer the accumulator contents into X and Y registers respectively and TXA, TYA transfer the Index register contents to the accumulator.

In some programs it may be necessary to have a computed address, rather than a base address with an offset, as in absolute indexed addressing. This is done using indirect addressing, instructions in this mode have just a single eight bit address field which points to the effective address as two bytes in page zero. The data address is thus not stored directly in the Operand field of the instruction but, indirectly in page zero, all the indirect addresses are indexed except for the JMP instruction. Two modes of indirect addressing are possible, Indexed Indirect and Indirect Indexed Addressing.

Indexed Indirect addressing index register X is added to the Operand zero page address. This points to locations where the sixteen bit data address is stored. One of the major uses of this addressing mode is in retrieving data from a table or list of addresses, as in polling I/0 devices or performing string operations.

Indirect Indexed addressing the sixteen bit address pointer in page zero is first accessed then offset by the contents of index register Y to give the true data address. The location of the pointer is fixed, whereas in the indexed indirect mode it is variable being offset by the contents of index register X. Indirect indexed addressing combines the advantage of an address that can point anywhere in memory with the offset capability of the index register. It is a particularly powerful method of accessing the nth element of a table, providing the start address is stored in page zero.

DATA MODIFY INSTRUCTIONS

A small group of instructions are not associated with any particular processor register. They are classified as read/modify/write instructions. They all read data from a memory location or accumulator, modify it in a particular way and store the modified data back into memory or the accumulator. These instructions perform four different data modifications, shift, rotate, increment and decrement.

A shift instruction is one which takes the contents of the accumulator or a memory location and shifts all bits one bit to the left or right. An example is the LSR-Logical Right instruction, here the data in the accumulator or memory is moved one bit to the right, bit 0 is placed in the carry flag and bit seven set to zero. Similarly the ASL-Arithmetic Shift Left instruction moves the data one bit to the left, bit seven is stored in the carry flag and bit 0 set to zero. Repeated shifts in the same direction will eventually result in the entire byte being set to zero. Herein lies the difference between a shift and a rotate instruction. In a rotate instruction the contents of the carry flag is stored in the bit emptied by the shift, thus no data is lost in a rotate instruction. The ROL-Rotate Left instruction shifts the contents of the accumulator or addressed memory left 1 bit with the carry stored in bit 0 and bit 7 stored in the carry flag. With ROR-Rotate Right instruction the data is shifted right 1 bit with bit 0 shifted into the carry and the carry shifted into bit 7. The shift and rotate instructions have a unique form of addressing, in addition to the normal forms, and is known as accumulator mode addressing. It indicates that the instruction is to operate on the accumulator rather than on a memory location.

Besides shift and rotate the contents of a memory location can be incremented or decremented. INC-Increment Memory by One adds one to the contents of the addressed memory location. DEC-Decrement Memory by One subtracts one in two's compliment form from the contents of the addressed memory location. The main use of increment and decrement is with counters such as table pointers.

23

INTERRUPTS AND INITIALISATION

The processing of interrupts is important for the operation of the VIC system. As seen in Section 4 all peripheral 1/0 is interrupt driven, a knowledge of interrupts is thus required by anyone using the user port or the other 1/0. There are three input lines which can cause the processor to halt on completion of the current instruction. On receipt of one of these inputs the program counter is stored on the stack and the processor causes the program to jump to an interrupt servicing routine at an address pointed to by the contents of one of the interrupt vectors. These three lines are Reset, Interrupt Request or IRQ, and Non-Maskable Interrupt or NMI. All three lines can be used by external devices attached to the VIC memory expansion port. Their function can be controlled by the programmer thanks to the RAM vectored address table which allows user written routines to replace the system routines governing interrupts and reset.

The only way a programmer can change the sequence of operations is to load a new address into the program counter. If this were true then an external event could not effect the program sequence, unless the program was written to periodically check for an input. Most inputs are asynchronous, meaning that for an input to occur at the same time as the program is checking for inputs is extremely unlikely. If an input pulse occurred just after an input check, then not until the next check would that pulse be input to the computer. During the interval between checks; data at the input may have changed resulting in the loss of information. To overcome such a data loss the processor could be programmed to wait for the data, but this would mean the processor spending most of its time doing nothing.

The problem of having the processor wait for an input is overcome by having a special line signal the processor whenever an input occurs, an interrupt. This considerably simplifies programming, making it unnecessary to repeatedly use an input testing subroutine or have the computer wait for an input. The two interrupt lines used to signal to the processor that an input is present are the IRQ line and the NMI line. By pulling an interrupt line low for at least 20 microseconds an input device can signal that it wishes to send data to the processor. This forces the processor to finish its current instruction, store the program counter and status register on the stack and jump to a memory location pointed to by the interrupt vector. There are two interrupt vectors that for the IRQ line are located at 65,534 and 65,535,

for the NMI line at 65,530 and 65.531. The reset vector is located at 65,532 and 65,533.

The processor could be interrupted before it was able to retrieve data from an interrupt initiated input. To prevent this the programmer can disable the IRQ line and prevent further interrupts by setting the I flag in the processor status register. This is done by the first instruction in the interrupt handling subroutine, SEI-Set Interrupt Disable. A CLI — Clear Interrupt Disable instruction clears the I flag and allows the processor to be interrupted as normal. Having obtained data from the input the interrupt software can process it for use by the main program or respond with an output from an I/O port. Control is returned to the main program by the RTI-Return from Interrupt instruction. This pulls the contents of the processor status register and program counter off the stack restoring the processor to its pre-interrupt state. The NMI line can not be disabled with commands to the processor and will therefore always generate an interrupt irrespective of the state of the IRQ line and the Status register I flag. An interrupt on the NMI line therefore has a higher priority than an input on the IRQ line. The Reset line takes priority over both the interrupts causing all the system pointers to be reset and a Basic warm start initiated. If the Reset RAM vector address is changed then the user written reset routine which it points to must clear the processor registers, and reset the stack pointer to the beginning of the stack, before jumping to the entry point of the main program.

The VIC has two sources of interrupt, one from each of the peripheral 1/0 chips and either one can interrupt the processor. The interrupt line from one I/0 chip (VIA No.2) is connected to the IRQ line (a timer on this chip generates regular interrupts which control update of the clock variable TI and keyboard scanning). The interrupt from the other I/0 chip (VIA No.1) is connected to the NMI line and is used to generate a system restart when the Restore key is pressed. Each I/0 chip has two interrupt inputs and one output connected to the IRQ or NMI lines. The function of these I/0 inputs is dealt with in the sections on the 6522 VIA chip and the system I/0.

An interrupt sequence can be created by the programmer without an input being present in the IRQ line, by use of the BRK — Break command. This instruction performs a software interrupt and causes program control to be transferred to the address stored in the interrupt vector. The main use of this instruction is in debugging a program, however, it calls one of the interrupt routines. Its use on the

VIC is not recommended. For VIC users a similar function is provided in the machine code monitor with none of the attendant problems of the BRK instruction.

MACHINE CODE ON THE VIC

The VIC has an advantage over many other small micro computer systems in that it can be programed in both Basic and machine code. this gives the programmer the powerful option of using machine code subroutines in a Basic program. The VIC normally runs in the Basic mode and there are six ways of accessing the machine code environment. The first two use commands in Basic, these are, USR and SYS. Both commands access a machine code subroutine whose address is specified in the command or in a specific page zero location. The next four methods involve adding machine code subroutines into the operating system.

1 — the Basic command USR(X) transfers program control to an address stored in locations 1 and 2, this address is user definable and will be the start of a machine code subroutine. The value X specified in the command is a parameter for use by the subroutine, this is evaluated and placed in floating accumulator No.1 starting at location $0061. A parameter may be returned by placing it in the floating accumulator and providing it is in the correct format then this value will be assigned to the parameter variable.

2 — the Basic command SYS(X) causes program control to jump to a machine code subroutine starting at address location X, where X is either a variable or a constant value equal to the decimal start address. Parameters can be passed between the Basic program and the machine code routine using POKE and PEEK commands to place or read values from specified memory locations.

3 — if the machine code routines are located in ROM memory and start at memory address No.A000 then the VIC allows system control to jump to this location rather than the normal Basic interpreter when the machine is switched on. This is very useful since it allows the user to change the VIC system. This can be either adding extra commands to Basic, changing the I/0 operation using the ROM and RAM jump vectors or simply bypassing the Basic and operating system software and replacing it with special custom software (this is commonly done for cartridge games).

4 — add a program into the interrupt servicing routines, these are called sixty times a second by the keyboard scan interrupt signal. This method for example allows the scanning of I/0 ports for an input, or selectively disabling certain keys on the keyboard. Any situation

27

where a program must be run concurrently with the main program could use this method.

5 — involves inserting extra code into the CHARGOT subroutine which gets each line of Basic from memory prior to its execution by the interpreter. By intercepting each line of Basic before it is executed new Basic instructions can be added. The instruction being performed by a user written machine code subroutine. Both the method of inserting code into the interrupt routine and the addition of extra code into the CHARGOT subroutine will be dealt later with in full.

6 — the RAM vector address table can be used to insert code into, or replace, any one of the Basic or system routines accessible through this table. This is similar to method 4 and could be used to reassign the functions of the interrupt lines or change the peripheral I/O handling routines.

The main reason for using machine code subroutines is that Basic is too slow for many purposes, especially when using the I/O ports or in special purpose display functions. A machine code routine is more than 100 times faster than the same program written in Basic. Another reason for using machine code is that one may want to change the operating system or use some of the operating system subroutines.

A machine code program which is loaded into RAM memory is best located at the top of memory. This area is used by Basic to store character strings, and to avoid these overwriting the machine code program the top of memory pointers must be changed. The top of memory pointers are set during power up diagnostics to the highest usable RAM location. The location of the top of memory and therefore the values stored in the top of memory pointer bytes depends on whether the VIC is fitted with any extension RAM. By lowering the value of these pointers a block of memory can be reserved exclusively for use by a machine code program. The operating system will regard the new top of memory pointers as containing the highest memory location usable by Basic. The pointer is stored as the low order byte in 51 and the high order byte in 52. As an example the following commands will lower the top of memory to location 4096:
```
POKE 51,0 : POKE 52,16
POKE 55,0 : POKE 56,16
CLR
```

Locations 55 and 56 are the top of strings pointers and must be set equal to the top of memory pointers at the start of the program, the CLR command resets all the variable pointers thereby clearing all variables used previously in the program. Care should be taken when locating machine code programs in RAM memory space that the memory area used is not also allocated to either video memory or character generator memory.

Of the two Basic commands used to call a machine code subroutine, SYS and USR, by far the most powerful and flexible is SYS. With the SYS command one simply specifies the subroutine starting location, thus if it starts at location 5000 it can be called with SYS 5000. Variables can be transferred between a Basic program and a machine code program by using PEEK and POKE. These read or write single or multiple byte values into memory locations allocated for the purpose and accessed by both programs. Transferring variables in this manner is easier than using the single floating point variable provided for the USR function. It also allows the transfer of more than one variable which USR does not. The only requirement with a SYS subroutine is that the last instruction in the subroutine is a RTS — return from subroutine, this automatically returns control to the Basic program.

The easiest way of entering a machine code program is to incorporate it into the Basic program using a simple loader, on running the program the loader POKES the values byte by byte into the correct locations. Another way is to use the machine code monitor which is part of the Programmers Aid ROM pack, a summary of all the commands in the monitor are given in Appendix 6. The monitor allows machine code program to be directly written into memory using hexadecimal code. It also allows programs to be saved and loaded onto tape in machine code format. To make the writing of machine code programs easier and avoid the necessity of hand encoding, a simple assembler and disassembler are included in the monitor. The monitor saves a machine code program by saving the block of memory where the program is located, far quicker than a corresponding Basic loader.

The only drawback with using the monitor to save and load a machine code program is that it will require a two part load, the first to load the machine code and the second to load the Basic program calling the machine code routine. The Basic program could be saved by the monitor together with the machine code, by saving the entire contents of user memory from location No. $0400 up. Generally it is

```
10 REM ***********************************
20 REM *BASIC LOADER FOR MACHINE CODE
30 REM *ROUTINE (EXAMPLE CODE ONLY)
40 REM ***********************************
100 DATA 2048: REM **CODE START LOCATION
105 REM **MACHINE CODE IN HEXADECIMAL
110 DATA 48,98,48,8A,48
120 DATA A9,13,20,D8,E3,A5,54,F0,09
130 DATA A9,11,20,D8,E3,C6,54,D0,F7
140 DATA A5,55,F0,09,A9,1D,20,D8
150 DATA 68,AA,68,A8,68,60
9000 DATA*: REM **END OF CODE
9005 REM **THE FOLLOWING LINES ARE THE BASIC
9006 REM **LOADER PROGRAM.
9010 READL
9020 READA$
9030 C=LEN(A$)
9040 IFA$="*"THEN9140
9050 IFC<1ORC>2THEN9130
9060 A=ASC(A$)-48
9070 B=ASC(RIGHT$(A$,1))-48
9080 N=B+7*(B>9)-(C=2)*(16*(A+7*(A>9)))
9090 IFN<0ORN>255THEN9130
9100 POKEL,N
9110 L=L+1
9120 GOTO9020
9130 PRINT"BYTE"L"=["A$"] ????"
9140 END
READY.
```

best to use a Basic loader for short machine code programs which are called by the main Basic program. Longer machine code subroutines and machine code programs which stand alone and are not called from Basic are best saved using the monitor.

Another method of storing machine code programs is to store them within a Basic program as REM statements. To do this the machine code program must first be split into blocks, each block being less than 80 bytes long. Each byte of the machine code routines is stored as a character in the REM statements. The REM statements are stored as the first few lines of the Basic program. Each statement is first filled with dummy characters, the number of characters in each statement depends on the length of the block of machine code to be stored in that statement. The machine code monitor is then used to find each REM statement as it is stored in memory and replace the dummy characters with the code value for each byte in the machine code block. When the program is listed the REM statements will appear as a seemingly random collection of ASCII characters, each character however represents a byte of the machine code routine. When writing a machine code program to be stored in this way care should be taken to ensure that no absolute jump addresses are used within a block, this ensures that the routine is relocatable. Care should be taken to ensure that jump addresses from the main calling routine are suitably modified to allow for the six byte gaps in the program required for storage of line number, link address, command token and terminating 0, and the location of each routine.

WRITING MACHINE CODE PROGRAMS

The prospect of writing a machine code program even a small one may seem fairly daunting, but providing one uses an orderly and disciplined approach to the problem it need not be difficult. A machine code program differs from Basic in the approach taken to its writing. Whereas a rough Basic program can be written and then polished up by inserting and changing lines, a machine code program must be written as the final version, any changes often necessitate rewriting and assembling the whole program. Machine code unlike Basic code is dependent on the exact position of instructions in memory. Adding a couple of instructions into the middle of a program means changing most jump, branch and data addresses. Machine code programs also require far greater attention to details like current flag status, programs must be very carefully planned before they are written. Unless this is done, writing a machine code program will require far greater effort than is necessary and the product far more prone to error. It is strongly recommended that before writing any programs in machine code yourself, you study some 6502 machine code routines, try to determine why the code was written in a particular way and what it does.

Stage one in planning a program is to define what the program is required to do, then break the problem into a series of steps. To demonstrate this consider the following example, to display all the ASCII characters on the screen.

Set the screen location pointer LOC to start of screen, address 32768 — set the ASCII character value CHAR to zero — store character code CHAR on screen at location LOC — increment LOC — increment CHAR — if CHAR is greater than 255 then all characters have been displayed and program ends, if not then go back and display next CHAR.

From this description we have defined that two variables CHAR and LOC are required, also the program structure requires a loop with a conditional test. For a short program like this a written description is not really required since one can easily remember what one wants the program to do. For longer programs it is an essential part of the process. From the written description one can construct a flow diagram such as the example in Figure 5. The flow diagram can be regarded as a pictorial version of the written description and as a result simpler to follow.
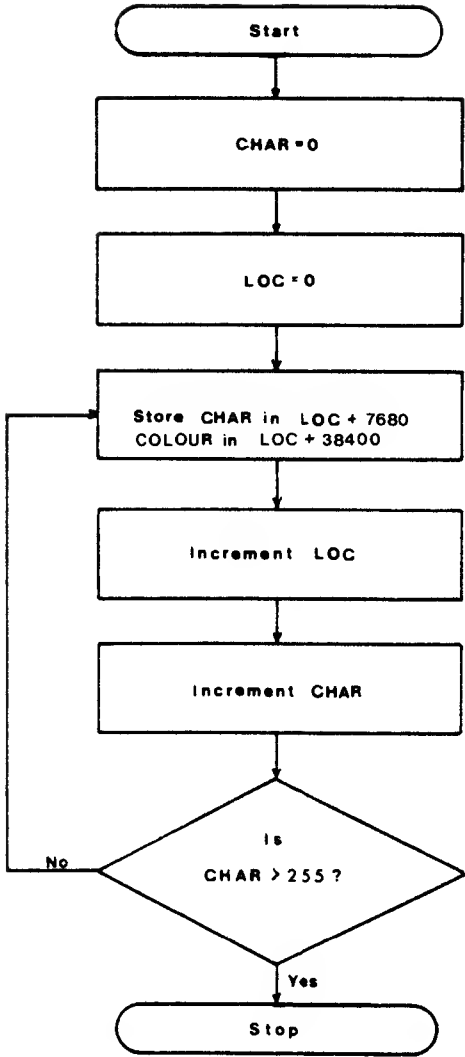
Fig. 5 — Preliminary flow diagram for example routine.

For long programs the flow diagram and written description can get very involved and confusing. It is good practice to split such a program into a series of self contained blocks or subroutine modules. Each module is then treated as a complete program, making program writing and debugging easier. The flow diagram shows the logical pathways through a program and most logical errors can usually be detected at this stage, saving a considerable amount of programming time.

Having drawn a flow diagram the next stage is the construction of a table of variables and locations of system subroutines called. In the example no system subroutines are used but two variables are required:
LOC — pointer to location in screen memory where character is to be stored.
CHAR — Value for ASCIII character to be displayed on screen.
It is important that the table contains all variables required, since when writing the program exactly the right amount of space in memory must be left to contain them.

Having defined the logical flow of the program, the variables used and any system subroutines called, a start can be made on writing the program code. Probably the best way is first to draw an expanded version of the flow diagram. Breaking down each logical step into a series of substeps corresponding to a machine code instruction. In Figure 6 notice that the variable LOC is now stored as the contents of the X index register. Indexed addressing being the easiest way of putting data into successive memory locations. Also the index register (i.e. LOC) is loaded with 255 and decremented, rather than 0 and incremented as in the original flow diagram, since it is easier to test for zero than for 255. It should be noted that the coding of this example is not the optimum for either speed or compactness but rather for clarity so that the purpose of each command is clearly visible.

Having laid out the program in flow diagram form the next step is to write the actual code which will be used by the machine. There are three ways in which this can be done, the choice of which method is used depends on equipment available and the size of the program. The first method is to write the code by hand using a coding form using the instruction mnemonic or operand address/value in the opcode column, see Figure 7. Code written and assembled by hand can best be entered into the VIC memory by a Basic loader program.
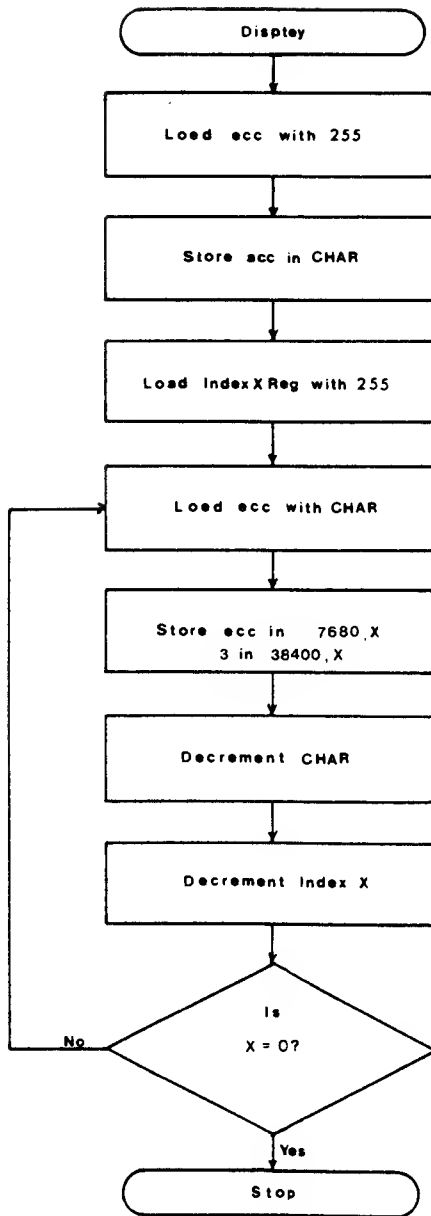
Fig. 6 — Expanded flow diagram for example routine.

The second method again involves writing the code in mnemonic and label form by hand on a coding form, the assembler which is part of the monitor on the VIC Programming Cartridge is used to assemble the code and directly enter it into the VIC memory. The third method is best used for very long machine code programs and involves using a PET computer system (a 32K minimum with disks and printer) on which to write and develop the code. The code is written using the editor program in the assembler package, this creates a source code file on disk, the assembler uses this file during assembly to create an opcode file on disk which can be loaded into memory using the loader routine.

For hand assembly and coding of a program it is advisable to use a coding form such as that shown in the example, it helps to considerably reduce the number of errors occuring at this stage. On the first page of the coding form should be written a list of all variables, I/0 locations and system subroutine entry points used. Each variable being assigned the number of bytes of memory which it will require. Most will be single byte but some will be two or three byte precision and in the case of character variables or data buffers memory required could be large. When storing a multiple byte numerical variable it is good practice to store the bytes in fixed order, with the least significant byte in the first location and the most significant byte in the last location. It is easier this way to keep track of which part of a variable is being dealt with. Also index registers can be used to access successive bytes of a variable in the same order that they are processed.

Program variables can be stored in any part of RAM memory not occupied by either programs or system variables. For maximum speed and reduced program size variables should be stored in page zero of memory, the bottom 255 bytes. On the VIC page zero is currently occupied by system variables. This area can be utilised if the memory locations used are carefully chosen. If Basic is not used, then the entire section of page zero used by the Basic interpreter for variable storage (locations 0 to 143) is available to the programmer. The remaining part of page zero is used by the operating system and may or may not be required by the machine code program. If both Basic and machine code are to be used together in the same program then the number of page zero locations available is limited (locations

ADDRESS MSB	LSB	OPCODE	LABEL	MNEMONIC	AD MODE	OPERAND	FLAGS Z	N	C	I	D	V	CYCLE	COMMENT
03	4 0	—	CHAR	—										VARIABLE FOR ASCII CHARACTER
	1	A9	DISPLAY	LDA	#	255								START — SET UP LOOP COUNT
	2	FF		/										AND CHARACTER VALUE
	3	8D		STA	ABS	CHAR								INITIALISE 'CHAR'
	4	40		/										
	5	03		/										
	6	A2		LDX	#	255								SET INDEX REG = 255
	7	FF		/										
	8	AD	NEXTCHAR	LDA	ABS	CHAR								GET 'CHAR'
	9	40		/										
	A	03		/										
	B	9D		STA	ABS,X	$1E00,X								STORE AT START OF VIDEO RAM
	C	00		/										+ INDEX
	D	1E		/										
	E	A9		LDA	#	03								SET COLOUR = RED
	F	03		/										
	5 0	9D		STA	ABS,X	$9600,X								STORE COLOUR IN COLOUR RAM
	1	00		/										AT $9600,X
	2	96		/										
	3	CE		DEC	ABS	CHAR								PUT NEXT ASCII CHARACTER
	4	40		/										IN 'CHAR'
	5	03		/										
	6	CA		DEX	IMP		X							POINT TO NEXT SCREEN
	7	D0		BNE	REL	NEXTCHAR								LOCATION — LAST CHARACTER?
	8	EF		/										
	9	60		RTS	IMP									END AND RETURN FROM SUBROUTINE
	A													
	B													
	C													
	D													
	E													
	F													

Fig. 7 — Hand coded program of example routine (note this is the simplest though not necessarily the best way of writing this program).

37

87 to 96 are best). If a larger section of page zero memory is required then the existing contents should be relocated to a protected part of memory before the machine code routine is run and restored at the end of the routine.

Using the second expanded flow diagram one can start writing the code onto the coding form using the instruction mnemonics. The first step is to enter the starting location of the program into the address column, then enter the first instruction into the mnemonic column. The addressing mode of the instruction should be entered into the relevant column. This is important since one must be able to calculate how many bytes are required by that instruction, to determine on which line (i.e. at which address) the next instruction should be entered. The label column will contain an entry only if that address is the start of a subroutine or the destination of a jump or branch instruction. On the flow diagram the position of labels is indicated where an operation has more than one entry or exit point. The label used can be any name but preferably one descriptive of the function of the subroutine or loop. In the example the beginning of the program is given the label DISPLAY and the entry point of the loop is called NEXTCHAR. Entries in the operand column will only be required for instructions referencing other locations in the program and will consist of symbolic labels and variable names. As program code is entered on the coding form the comment column should also be completed. Either with simple references to the flow diagram or a more complete description. At a later date the function and logical flow of the program can thus be easily followed without relying on memory.

If the machine code routine is to be called from a Basic program with either a SYS or USR command then it is very important that the contents of the processor registers are saved before the routine is executed and then restored at the end of the routine. This is most easily done using the stack. The first few instructions of the routine push all registers onto the stack and the last instructions restore register contents by pulling the correct values off the stack.

Once written, the program should be checked for logical errors, before being assembled. It will involve less work if errors are detected prior to assembly. Assembly of short to medium length programs is, in the absence of a full assembler running on a PET, best done with the

spot assembler function of the monitor. Full details of the monitor functions are given in Appendix 6. The process of hand assembling is done, in the absence of a monitor, in two stages, the first consists of using the instruction set list to obtain the opcode value for each mnemonic with the specified addressing mode. This hexadecimal value is entered into the opcode column of the coding form on the same line as the mnemonic. If the addressing mode is other than "implied" or "acumulator" then the following one or two bytes will be used to store an address or a value specified in the operand column. If the addressing mode is immediate, then the operand column contains a hexadecimal value which is transferred to the opcode column on the line following that of the instruction code.

The number system used must always be noted, the conventions are that a number prefixed with a % is in binary format, with a $ in hexadecimal format and if no prefix is given then in decimal format. Convention also dictates that an instruction in the immediate mode is identified by a No. sign in the address mode column, all other address modes are just an abbreviation of the name. For all other modes the symbol contained in the operand column will correspond to either a label or variable. If a variable, then the address of the variable can be obtained from the variable table on the first page of the coding form. If the instruction is a jump or branch then the addressing mode used will transfer program control to another section of the program, the operand column will thus contain a label. Since a label needs the calculation of a jump address it is left until the second part of the assembly procedure. It should be noted that the 6502 requires that all addresses are stored in the form "least significant byte" first, then "most significant byte" thus address 0340 hexadecimal is stored as 4003.

At the end of the first stage of the assembly process, the opcode column on the coding forms should contain a list of hexadecimal values, one for each location in memory. The exceptions being jump and branch addresses which are calculated in the second stage. Jump addresses pose no problem since they are stored in either indirect or more commonly absolute mode. Their entries in the opcode column can be obtained from the address of the relevant label. The conditional branch instructions all use relative addressing, where the branch, either forward or backward, is calculated from the location of the branch instruction rather than a fixed location in memory. It is the offset from the current location, which can be up to 127 bytes away,

either forward or backward, which must be calculated by the programmer. Great care should be taken with this, any error will cause program control to be transferred to the wrong place, with resultant errors or program crash. To calculate the value for a forward branch one counts the number of bytes from the location of the branch instruction, to the location of the label in the branch operand column, and subtract 2 from this value. If the branch is backwards then the offset is calculated by counting the number of bytes from the branch instruction to the label, then adding 1 and subtracting from 255. The result when converted into hexadecimal can be stored in the opcode column after the branch instruction.

Once all jump addresses have been calculated and a complete list of opcode values obtained the program can be entered into the computer. Before this is done it is advisable to recheck the program, especially the opcode listing for errors (make sure that you can distinguish between 8 and B or A and 4). The opcode listing is then entered into the VIC using either a Basic loader or the machine code monitor. Once entered, the program should be saved before it is run since it is very rarely that a machine code program runs perfectly first time. With the aid of the monitor the contents or memory should be checked against the opcode listing for any program entry errors. If any are found they should be corrected and the program resaved. One can then try running it. If there is a program error it will probably crash the machine, if so reload the program and the monitor and carefully recheck the logic flow, the coding and the contents of memory. In my experience the three most common causes of fatal program errors are — entry errors, coding errors, and wrongly calculated jump and branch addresses.

The best way of detecting errors is to systematically work through the program inserting a break instruction at points where program failure may have occurred. This will cause the program to return to the monitor, allowing the contents of variable locations to be checked and gradually isolate the fault to a small section of code. Another way of isolating errors is to run the program from different locations, though this does require a careful choice of entry points. Having detected and removed any fatal errors one may find that the program still does not run properly and produces strange results. Non fatal errors are most commonly caused by either a mistake in the basic logic flow, ignoring the current flag status, using the wrong variable, and quite commonly using the wrong branch instruction.

Successful machine code programming is not difficult, it requires just a strict adherance to a method and constant attention to detail plus plenty of practice. The methods outlined above should enable VIC users to expand their machine's capabilities by using machine code subroutines.

2 VIC SYSTEM SOFTWARE
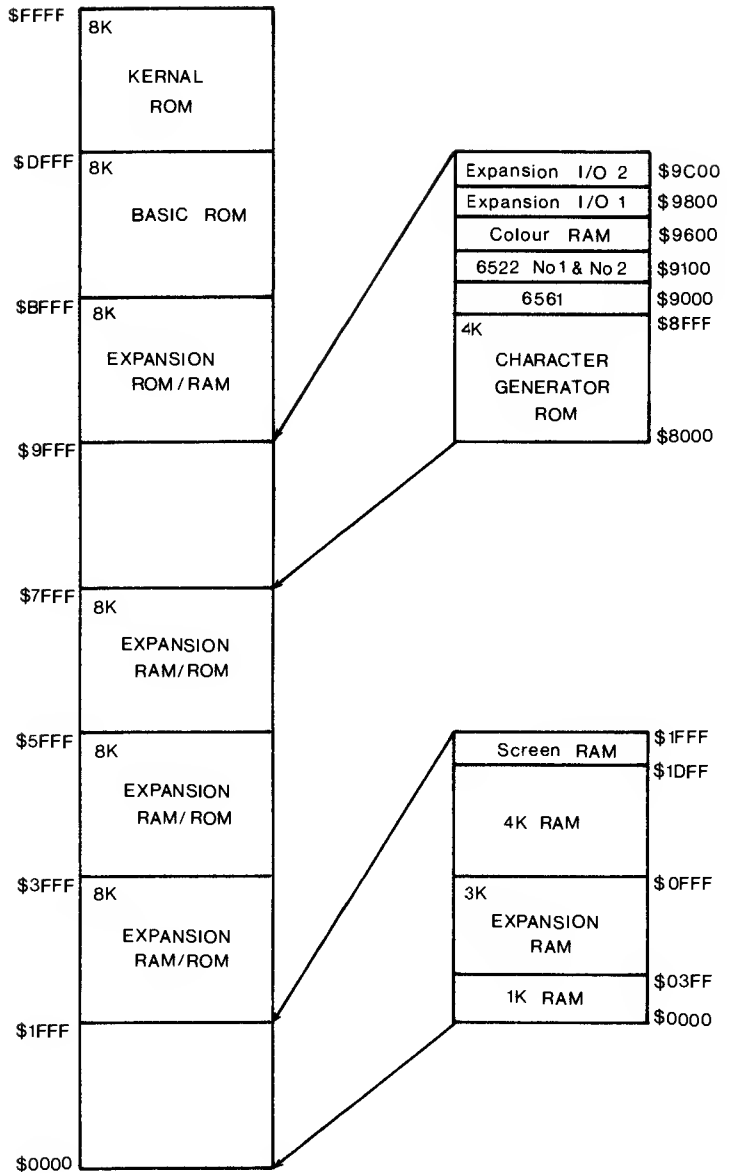
Fig. 8 — VIC memory map.

VIC MEMORY MAP

The 6502 microprocessor used in the VIC is capable of accessing up to 64K bytes of memory, this memory space is divided into blocks having a specific function. On the basic VIC only 29K of the available 64K is utilised, the remainder is available for user expansion using either ROM or RAM memory or even specialised I/O, all accessible through the memory expansion connector at the rear of the machine. The division of the memory space into blocks with different functions is shown in Figure 8. An understanding of the function and location of each block is essential if full use is to be made of the VIC.

1 — System variables — Hex $0000 to $03FF — Decimal 0 to 1023 — The first 1024 bytes of RAM memory are utilised by Basic and the operating system for the storage of system variables. The VIC system configuration and/or its mode of operation can be changed by placing values into specific locations in this section of memory.

2 — User RAM memory — Hex $0400 to $7FFF — Decimal 1024 to 32,767 — This 31K block of addressable memory can be divided into four sub sections the first of 7K length and the remaining three sections of 8K length. The first sub section consists exclusively of RAM memory it is made up from the VIC's built-in 4K user memory which extends from location $1000 to $3FFF plus the standard 3K expansion RAM which goes from location $0400 to $0FFF. If there is no other RAM memory expansion then the top 512 bytes of the first section of RAM memory $1EOO to $1EFFF are used as the screen memory (if the High Res mode is used then in addition to the screen memory a 4K block of RAM is required for the programmable character generator, see the section on the 6561 for details). If there is more than 7K of user RAM in the system then the screen memory is moved to start at $1000, decimal 4096. The three 8K sections of user memory can be either RAM or ROM and are completely free for user programs and data with the exception of the screen memory.

3 — Character Generator — Hex $8000 to $8FFF — Decimal 32,768 to 36,863 — The character generator is a 4K ROM which contains the pattern of dots used to display each of the 255 valid VIC ASCII characters on the screen. The contents of the character generator will depend on which language version of the VIC you possess, there are (at the time of writing) three versions. The programmer does not need to bother about the character generator in normal character display

mode. However, in High Resolution display mode the character generator is not used and therefore to display alphanumeric characters in this mode the data for the desired character must be transfered from the character generator to the block of RAM used for the user definable character generator.

4 — System I/O and Control Interfaces — Hex $9000 to $912F — Decimal 36,864 to 37,167 — All the system input, output and control lines are memory mapped, this means that an I/O line can be turned on or off simply by changing the corresponding bit in a specific memory location. The internal registers of three I/O chips are accessable within this block of memory, they are two 6522 VIA chips and the 5621 VIC chip, the latter controls the operation of the video display. An understanding of the operation of these three chips is essential if any of the many interfaces between the VIC and external devices are to be used, consequently two complete sections of this book are devoted to these devices.

5 — Colour Memory — Hex $9400 to $95FF — Decimal 37,888 to 38,399 — Each of the 506 bytes in this block of memory determines the foreground and background colour of the corresponding byte in the video memory. It should be noted that if there is more than 7K of user RAM in the system then the colour memory starting address is moved up to $9600, decimal 38,400.

6 — ROM Expansion Memory — Hex $A000 to $BFFF — Decimal 40,960 to 49,151 — This 8K block of memory is designed for use by programs stored in ROM and contained in a ROM/RAM pack plugged into the VIC memory expansion port. The VIC operating system allows a machine code program, starting at location $A000, to power up directly into that program on switching on the machine, rather than into Basic.

7 — VIC Basic Interpreter — Hex $C000 to $DFFF — Decimal 49,152 to 57,343 — The interpreter translates a high level Basic program, step by step into a series of machine code routines, these perform the function required for each Basic command.

8 — VIC Operating System Kernal — Hex $E000 to $FFFF — Decimal 57,344 to 65,535 — The operating system controls the functioning of the VIC system, such as initialisation on power up, communication with peripheral devices, screen display and editing

45

etc. The operating system normally works in conjunction with the Basic interpreter but the routines within it can be used by any machine code program requiring the operating system functions.

VIC SYSTEM VARIABLES

The entire block of memory from location 0 to 1023 is reserved for use by the VIC system software, it is used to contain system variables, temporary data storage, and input/output buffers. This memory area is accessable to the user via PEEK and POKE commands in Basic, or simple load and store commands in machine code (locations 256 to 511 contain the processor stack, this should not be accessed except by processor stack commands in machine code). User accessability of this area of memory is important since many interesting and useful operations can only be performed by reading or changing the contents of one or more locations within this bottom 1K of memory. The function of each location can be seen in Table 1, the programmer should study this very carefully before attempting to use or modify any of the variables. The memory area used by the VIC for variable storage can be divided into seven distinct sections each with a different function and used by a different part of the system software.

1 — Basic Interpreter Variables — Hex $0000 to $008F — Decimal 0 to 143 — This section of page zero is used exclusively by the Basic interpreter and of the variables stored in this section and 16 two byte pointers stored between location 43 and 74 are the most interesting. If machine code subroutines are being used in a Basic program then locations 87 to 96 can be used for page zero variable storage, if a machine code program is being run which does not require the Basic interpreter then the whole 143 bytes in this block may be used for variable storage. If the USR command is used or any of the Basic function routines are called from a machine code program then the Floating Accumulators 97 — 112 will be required for transfers of variables.

2 — Operating System Kernal Variables — Hex $90 to $FF — Decimal 144 to 255 — All the variables and parameters stored in this last section of page zero are of interest to the programmer. They control the input and output functions of the VIC on the RS232 and IEEE port, the allocation of files for I/O with peripheral devices, data transfer between the VIC and the cassette deck, the control of the screen editor, etc.

3 — Processor Stack — Hex $0100 to $01FF — Decimal 256 to 511 — The section of memory occupied by the stack is common to all 6502 processor systems. The processor uses the stack as a last in first out

47

buffer to store temporary data, such as return addresses in subroutine calls. The operation of the stack is automatically controlled by the processor and this area of memory should only be accessed with extreme caution.

4 — Basic Buffer — Hex $0200 to $0258 — Decimal 512 to 600 — These 89 bytes are used to temporarily store text or program lines (a line is four screen lines maximum length or 88 characters) during input or output operations. When a program line is input it is transferred from the Basic input buffer to memory by the terminating carriage return. The line is then converted into the form in which it is stored in memory, all the commands are converted into tokens thus reducing the memory space occupied by the program. This area of memory should be accessed with extreme caution.

5 — Operating System Kernal Variables — Hex $0259 to $02FF — Decimal 601 to 767 — These operating system parameters and variables do not require to be stored in page zero locations, they are a continuation of the variables stored in page zero and deal with the same functions. All locations in this section can be usefully accessed.

6 — Indirect Addressing and Vectored Jumps — Hex $0300 to $0334 — Decimal 768 to 820 — This section of memory is used to store indirect jump addresses for system functions and is thus of considerable interest to the programmer. Some of the indirect addresses are only temporarily stored here and are not of great use to the programmer, 16 addresses are permanently stored here and they relate to most of the major operating system functions. These operating system vector addresses can be used to access the routines or to intercept the routines and either insert some extra code or replace the routine entirely. The programmer may want to intercept or replace an operating system routine for a variety of reasons. By inserting code into the interrupt routine which scans the keyboard and updates the screen and clock 60 times a second the programmer can, for example, get the VIC to automatically check the user port for inputs. The programmer may wish to completely change the operating system routines. If a non standard peripheral is connected to the VIC then the input and output routines could be changed. The use of vectored jump addresses increases the flexibility of the VIC system and allows the user to re-define the system to fit a particular situation.

7 — Cassette Buffer — Hex $033C to $03FC — Decimal 828 to 1020 — This 193 byte buffer is used in data transfers between the VIC and the cassette deck to store each block of data. If the cassette deck is not being used then this section of memory can be used to store small machine code programs.

HEX	DECIMAL	FUNCTION
$0000—$0002	0—2	USR function jump
$0003—$0004	3—4	Convert float — > integer
$0005—$0006	5—6	Convert integer — > float
$0007	7	General counter for Basic. Search character ':' or endline
$0008	8	Scan between quotes flag, 00 as delimeter
$0009	9	Column position of cursor on line (0—87)
$000A	10	Verify flag
$000B	11	Basic input buffer pointer;# subscripts
$000C	12	DIM flag. First character of array name
$000D	13	Variable flag, type: FF=string, 00=numeric
$000E	14	Integer flag, type: 80=integer, 00=floating point
$000F	15	DATA scan flag; LIST quote flag; memory flag
$0010	16	Subscript flag; FNx flag
$0011	17	Flags for input or read, 0=input, 64 = get, 1 = read
$0012	18	ATN sign flag: comparison evaluation flag
$0013	19	Current I/O device for prompt suppress
$0014—$0015	20—21	Basic integer address (for SYS, GOTO etc)
$0016	22	Temporary string descriptor stack pointer
$0017—$0018	23—24	Last temporary string vector
$0019—$0021	25—33	Stack of descriptors for temporary strings
$0022—$0023	34—35	Pointer for number transfer
$0024—$0025	36—37	Misc. number pointer
$002B—$002C	43—44	Pointer to start of Basic
$002D—$002E	45—46	Pointer to end of program start of variables
$002F—$0030	47—48	Pointer to end of variables start of arrays
$0031—$0032	49—50	Pointer to end of arrays
$0033—$0034	51—52	Pointer to start of active string space (coming down)
$0035—$0036	53—54	Pointer to top of active strings
$0037—$0038	55—56	Pointer to end of memory
$0039—$003A	57—58	Current Basic line number
$003B—$003C	59—60	Previous Basic line number
$003D—$003E	61—62	Pointer to Basic statement (for CONT)
$003F—$0040	63—64	Line number, current DATA line
$0041—$0042	65—66	Pointer to current DATA item

$0043—$0044	67—68	Input vector
$0045—$0046	69—70	Current variable name
$0047—$0048	71—72	Current variable address
$0049—$004A	73—74	Variable pointer for FOR/NEXT
$004B—$004C	75—76	Y save register-new operator save; current operator pointer
$004D	77	Special mask for current operator; comparison symbol
$004E—$004F	78—79	Misc. work area; function definition pointer hi-lo
$0050—$0051	80—81	Work area;pointer to string description
$0052	82	Length of above string
$0053	83	Constant used by garbage collect, 3 or 7
$0054—$0056	84—86	Jump vector for functions
$0057—$005B	87—91	Misc. numerical storage area
$005C—$0060	92—96	Misc. numberical storage area
$0061—$0066	97—102	Accumulator #1: E, M, M, M, M, S
$0067	103	Series evaluation constant pointer
$0068	104	Accumulator high order propagation word
$0069—$006E	105—110	Accumulator #2
$006F	111	Sign comparison, primary vs. secondary
$0070	112	Low order rounding byte for Acc# 1
$0071—$0072	113—114	Cassette buffer length/series pointer
$0073—$008A	115—138	Subrtn: Get Basic char; 7A, 7B = pointer (CHARGOT)
$008B—$008F	139—143	RND storage and work area
$0090	144	ST th I/O operation status flag
$0091	145	Stop key flag: Keyswitch pia.
$0092	146	Temporary
$0093	147	Load or verify flag
$0094	148	Cassette/IEEE load temp. IEEE buffered char. flag
$0095	149	IEEE 488 buffered character
$0096	150	Cassette sync #
$0097	151	Temp for IEEE input
$0098	152	How many open files; pointer to file table
$0099	153	Input device#, normally 0
$009A	154	Output CMD device, normally default of 3
$009B	155	Tape character parity
$009C	156	Cassette dipole switch
$009D	157	OS message flag, direct = $50, run = 0
$009E	158	Cassette error pass 1. Temporary

51

$0U9F	159	Cassette error pass 2. Temporary
$0UA0—$00A2	160—162	Jiffy clock
$0UA3	163	Serial bit count
$0UA4	164	Cycle counter for serial I/O
$0UA5	165	Countdown for tape write; sync on tape header
$0UA6	166	Cassette buffer pointer
$0UA7	167	RS-232 receiver input bit storage. Tape shortcount
$0UA8	168	RS-232 receiver bit count in. Tape read error
$0UA9	169	RS-232 receiver flag start bit check. Tape reading zeros
$0UAA	170	RS-232 receiver byte buffer. Tape read mode
$0UAB	171	RS-232 receiver parity storage. Tape short count
$0UAC—$00AD	172—173	Tape start address; tape buffer, scrolling
$0UAE—$00AF	174—175	Tape end address/end of current program
$0UB0	176	Temporary
$0UB1	177	Temporary
$0UB2—$00B3	178—179	Address of tape buffer#1 Y.
$0UB4	180	RS-232 transmitter bit count out
$0UB5	181	RS-232 transmitter next bit to be sent
$0UB6	182	RS-232 transmitter byte buffer
$00B7	183	Length of current file name string
$00B8	184	Current logical file number
$00B9	185	Current secondary address, or R/W command
$00BA	186	Current device number
$00BB—$00BC	187—188	Address of current file name string
$0UBD	189	RS-232 write shift word/Receive input character
$00BE	190	#blocks remaining to read/write
$00BF	191	Temporary
$00C0	192	Cassette manual/controlled switch
$00C1—$00C2	193—194	Tape start address (load)
$00C3—$00C4	195—196	Temporary
$00C5	197	Matrix co-ordinates of key down
$00C6	198	#of characters in keyboard buffer
$00C7	199	Reverse mode flag, 0 = off, 18 = on
$00C8	200	End of line for input pointer

$00C9—$00CA	201—202	Cursor log (row, column)
$00CB	203	Shift mode on print flag, which key, 64 if No. key
$00CC	204	Cursor blink enabled flag, 0 = on, 1 = off
$00CD	205	Delay before cursor blinks
$00CE	206	Character before cursor
$00CF	207	Cursor on/off blink flag
$00D0	208	Input from screen/input from keyboard
$00D1—$00D2	209—210	Screen address (row) pointer (screen memory)
$00D3	211	Position of cursor on current text line
$00D4	212	Quote mode flag, 0 = off, 1 = on
$00D5	213	Line length for screen (22, 44, 66, 88)
$00D6	214	Current screen line number
$00D7	215	Contain the ASCII value of last key press
$00D8	216	Insert mode flag
$00D9—$00F1	217—241	Screen line table: hi order address and line write
$00F2	242	Temporary for line index
$00F3—$00F4	243—244	Screen editor colour IP
$00F5—$00F6	245—246	Keyscan table indirect
$00F7—$00F8	247—248	Pointer to RS—232 receive buffer base location
$00F9—$00FA	249—250	Pointer to RS—232 transmitter buffer base location
$00FB—$00FF	251—255	Free kernal zero page locations
$0100—$010A	256—266	Floating to ASCII work area
$0100—$013E	256—318	Taps error log
$0100—$01FF	256—511	Processor stack area
$0200—$0258	512—600	Basic input buffer
$0259—$0262	601—610	Logical file number table
$0263—$026C	611—620	Device number table
$026D—$0276	621—630	Secondary address or R/W cmd, table
$0277—$0280	632—640	IRQ keyboard buffer
$0281—$0282	641—642	Start of memory
$0283—$0284	643—644	Top of memory
$0285	645	IEEE timeout flag
$0286	646	Active colour nibble
$0287	647	Original colour before cursor
$0288	648	Base location of screen (MSB)
$0289	649	Keyboard queue length
$028A	650	Repeat flag, 0 = cursor control only 255 = all keys

$028B	651	Delay before repeat occurs
$028C	652	Delay between repeats
$028D	653	Shift flag byte
$028E	654	Last shift pattern
$028F—$0290	655—656	Indirect for keyboard table setup
$0291	657	Shift mode switch, 0 = enabled, 1 = locked
$0292	658	Auto scroll down flag (0 = on,<> 0 = off)
$0293	659	6551 control register
$0294	660	6551 command register
$0295—$0296	661—662	Non standard (bit time/2-100)
$0297	663	RS-232 status register
$0298	664	Number of bits to send (fast response)
$0299—$029A	665—666	Baud rate full bit time
$029B	667	RS-232 receiver input buffer index to end
$029C	668	RS-232 receiver input buffer point to start
$029D	669	RS-232 transmitter output buffer index to start
$029E	670	RS-232 transmitter output buffer index to end
$029F—$02A0	671—672	Holds IRQ during tape operation
$02A1—$02FF	673—767	Free

BASIC INDIRECT JUMP ADDRESSES

$0300—$0301	768—769	Indirect error routine
$0302—$0303	770—771	Indirect main command handler
$0304—$0305	772—773	Indirect tokenisation routine
$0306—$0307	774—775	Indirect character list routine
$0308—$0309	776—777	Indirect character dispatch
$030A—$030B	778—779	Indirect symbol evaluation
$030C	780	Temporary storage during SYS of .A
$030D	781	Temporary storage during SYS of .X
$030E	782	Temporary storage during SYS of .Y
$030F	783	Temporary storage during SYS of .F

$0314—$0315	788—789	IRQ RAM vector
$0316—$0317	790—791	BRK instruction RAM vector
$0318—$0319	792—793	NMI RAM vector
$031A—$031B	794—795	Open logical file
$031C—$031D	796—797	Close logical file
$031F—$031F	798—799	Set input device
$0320—$0321	800—801	Set output device
$0322—$0323	802—803	Reset default I/O
$0324—$0325	804—805	Input from device
$0326—$0327	806—807	Output to device
$0328—$0329	808—809	Test STOP key
$032A—$032B	810—811	Get from keyboard
$032C—$032D	812—813	Close all files
$032E—$032F	814—815	Basic USR command vector
$0330—$0331	816—817	Load from device
$0332 — $0333	818—819	Save to device
$033C—$03FC	828—1020	Cassette buffer

0400—0FFF	1024—4095	3K expansion RAM area
1000—1DFF	4096—7679	User Basic area
1E00—1FFF	7680—8191	Screen memory
2000—3FFF	8192—16383	8K expansion RAM/ROM block 1
4000—5FFF	16384—24575	8K expansion RAM/ROM block 2
6000—7FFF	24576—32767	8K expansion RAM/ROM block 3

———————————————————————————————

NOTE: When additional memory is added to block 1 (and 2 and 3), the KERNAL relocates the following things for BASIC:

1000—11FF	4096—4607	Screen memory
1200—?	4608—?	User Basic area
9400—95FF	37888—38399	Colour RAM

———————————————————————————————

8000—8FFF	32768—36863	4K Character generator ROM
8000—83FF	32768—33791	Upper case and graphics
8400—87FF	33792—33815	Reversed upper case and graphics
8C00—8FFF	35840—36863	Reversed upper and lower case

9000—93FF	36864—37877	I/O BLOCK O
9000—900F	36864—35879	Address of VIC chip registers
9000	36864	bits 0—6 horizontal centering
		bit 7 sets interlace scan
9001	36865	vertical centering
9002	36866	bits 0—6 set No. of columns
		bit 7 is part of video matrix address
9003	36867	bits 1—6 set No. of rows
		bit 0 sets 8x8 or 16x8 chars
9004	36868	TV raster beam line
9005	36869	bits 0—3 start of character memory (default=)
		bits 4—7 is rest of video address (default=F)

BITS 3,2,1,0 CM starting address

		HEX	DEC
0000	ROM	8000	32768
0001		8400	33792
0010		8800	34816
0011		8C00	35840
1000	RAM	0000	0000
1001		xxxx	
1010		xxxx	unavail
1011		xxxx	
1100		1000	4096
1101		1400	5120
1110		1800	6144
1111		1C00	7168

9006	36870	horizontal position of light pen
9007	36871	vertical position of light pen
9008	36872	Digitized value of paddle X
9009	36873	Digitized value of paddle Y
900A	36874	Frequency for oscillator 1 (low) (on: 128—255)
900B	36875	Frequency for oscillator 2 (medium) (on: 128—255)
900C	36876	Frequency for oscillator 3 (high) (on: 128—255)
900D	36877	Frequency of noise source
900E	36878	bit 0—3 sets volume of all sound
		bits 4—7 are auxilary colour information

900F	36879	Screen and border colour register bits 4—7 select background colour bits 0—2 select border colour bit 3 selects inverted or normal mode			
9110—91FF	37136—37151	6522 PIA No. 1			
9110	37136	Port B output register (user port and RS232 lines)			

PIN ID	6522 ID	DESCRIPTION	EIA	ABV
C	PB)	Received data	(BB)	Sin
D	PBl	Request to Send	(CA)	RTS
E	PB2	Data terminal ready	(CD)	DTR
F	PB3	Ring indicator	(CE)	RI
H	PB4	Received line signal	(CF)	DCD
J	PB5	Unassigned	( )	XXX
K	PB6	Clear to send	(CB)	CTS
L	PB7	Data set ready	(CC)	DSR
B	CB1	Interrupt for Sin	(BB)	Sin
M	CB2	Transmitted data	(BA)	Sout
A	GND	Protective ground	(AA)	GND
N	GND	Signal ground	(AB)	GND

9111	37137	Port A output register (PA0) Bit 0=Serial CLK IN (PA1) Bit 1=Serial DATA IN (PA2) Bit 2=Joy 0 (PA3) Bit 3=Joy 1 (PA4) Bit 4=Joy 2 (PA5) Bit 5=Light pen/Fire button (PA6) Bit 6=Cassette switch sense (PA7) Bit 7=Serial ATN out
9112	37138	Data direction register B
9113	37139	Data direction register A
9114	37140	Timer 1 low byte
9115	37141	Timer 1 high byte 6 counter
9116	37142	Timer 1 low byte
9117	37143	Timer 1 high byte
9118	37144	Timer 2 low byte
9119	37145	Timer 2 high byte
911A	37146	Shift register

911B	37147	Auxiliary control register
911C	37148	Peripheral control register
		(CA1. CA2. CB1. CB2)
		CA1 = restore key (Bit 0)
		CA2 = cassette motor control
		(Bits 1-3)
		CB1 = interrupt signal for received
		RS232 data (Bit 4)
		CB2 = transmitted RS232 data
		(Bits 5-7)
911D	37149	Interrupt flag register
911E	37150	Interrupt enable register
911F	37151	Port A (Sense cassette switch)
9120—912F	37152—37167	6522 PIA No. 2
9120	37152	Port B output register
		keyboard column scan
		(PB3) Bit 3 = cassette write line
		(PB7) Bit 7 = Joy 3
9121	37153	Port A output register
		keyboard row scan
9122	37154	Data direction register B
9123	37155	Data direction register A
9124	37156	Timer 1. low byte latch
9125	37157	Timer 1. high byte latch
9126	37158	Timer 1. low byte counter
9127	37159	Timer 1. high byte counter
		timer 1 is used for the 60 time/
		second interrupt
9128	37160	Timer 2. low byte latch
9129	37161	Timer 2. high byte latch
912A	37162	Shift register
912B	37163	Auxiliary control register
912C	37164	Peripheral control register
		CA1 Cassette read line (Bit 0)
		CA2 Serial clock out (Bits 1-3)
		CB1 Serial SRQ IN (Bit 4)
		CB2 Serial data out (Bits 5-7)
912D	37165	Interrupt flag register
912D	37166	Interrupt enable register
912F	37167	Port A output register
9400—95FF	37888—38399	Location of COLOUR RAM with
		additional RAM at blk 1

9600—97FF	38400—38911	Normal location of COLOUR RAM
9800—9BFF	38912—39935	I/O block 2
9C00—9FFF	39936—40959	I/O block 3
A000—BFFF	40960—49152	8K decoded block for expansion ROM
C000—DFFF	49152—57343	8KBasic ROM
E000—FFFF	57344—65535	8K Kernal ROM

VIC USER MEMORY

The amount of memory available to the user depends on whether any RAM expansion cards are attached to the VIC, it will vary between 3K on a standard VIC to 31K on a fully expanded system. This memory space is however not completely available for program storage being also required for the storage of string and numeric variables and the screen memory. It is no use writing a program 3K long and trying to run it on a standard VIC as this will just result in the operating system giving an out of memory error. The Basic program is stored from location 4097 upwards (if the 3K RAM expansion card is fitted then programs start at location 1025) and the string and variables are stored from top of memory downwards.

Program Storage

When a program line is entered on the keyboard it is first written into the keyboard buffer. The operating system then transfers it byte by byte as it is entered onto the screen. The line however is not entered into memory until a carriage return is pressed. This causes the operating system to transfer the program line just entered from the screen into memory via the Basic buffer where the line of code is compressed and formatted. Each line is stored in a specific format using a compressed version of the Basic text. This reduces the memory requirements of a program and allows longer programs to be run. The compression of Basic text involves conversion of the Basic commands into single byte tokens. The command PRINT instead of being stored as five ASCII characters is stored in a single byte as the decimal value 153. When a program is listed the text compression process is reversed, as far as the user is concerned the program is stored in the same form as it was written.

A useful result of text compression is a shorthand way of writing Basic commands either in a program or direct command mode. This relies on the fact that the routine which converts commands to tokens looks only at the first two or three characters of a command word. Other characters in the command word are there for the users convenience only. Normally if we entered only the first couple of characters of a command the computer would respond with an error message. This can be done by using a simple method of fooling the error detection routines. Enter any Basic reserved word, type the first letter of the word, then depress the shift key and type the second letter. By using just the first two letters there could be confusion

VIC-20 BASIC Keyword Codes

Code (decimal)	Character/ Keyword	Code (decimal)	Character/ Keyword	Code (decimal)	Character/ Keyword	Code (decimal)	Character/ Keyword
0	End of line	66	B	133	INPUT	169	STEP
1-31	Unused	67	c	134	DIM	170	+
32	space	68	D	135	READ	171	—
33	!	69	E	136	LET	172	.
34	. .	70	F	137	GOTO	173	/
35	≠	71	G	138	RUN	174	↑
36	$	72	H	139	IF	175	AND
37	%	73	I	140	RESTORE	176	OR
38	&	74	J	141	GOSUB	177	>
39	'	75	K	142	RETURN	178	=
40	(	76	L	143	REM	179	<
41	)	77	M	144	STOP	180	SGN
42	.	78	N	145	ON	181	INT
43	+	79	O	146	WAIT	182	ABS
44	.	80	P	147	LOAD	183	USR
45	—	81	Q	148	SAVE	184	FRE
46	.	82	R	149	VERIFY	185	POS
47	/	83	S	150	DEF	186	SQR
48	0	84	T	151	POKE	187	RND
49	1	85	U	152	PRINT ≠	188	LOG
50	2	86	V	153	PRINT	!fl(	EXP
51	3	87	W	154	CONT	190	COS
52	4	88	X	155	LIST	191	SIN
53	5	89	Y	156	CLR	192	TAN
54	6	90	Z	157	CMD	193	ATN
55	7	91	[	158	SYS	194	PEEK
56	8	92	X	159	OPEN	195	LEN
57	9	93	]	160	CLOSE	196	STR$
58	:	94	↑	161	GET	197	VAL
59	;	95	←	162	NEW	198	ASC
60	<	96-127	Unused	163	TA8(	199	CHR$
61	=	128	END	164	TO	200	LEFT$
62	>	129	FOR	165	FN	201	RIGHT$
63	?	130	NEXT	166	SPC(	202	MID$
64		131	DATA	167	THEN	203-254	Unused
65	A	132	INPUT	168	NOT	255	π

Note that the left parenthesis is stored as part of the one-byte token for functions TAB and SPC, however, the other functions use a separate byte for this symbol. For example, the line:

10 IF INT(A) <5 THEN PRINT TAB(X)

would be coded as the following bytes (in decimal):

| Link | 10 | 0 | 139 | 32 | 181 | 40 | 65 | 41 | 179 | 53 | 32 | 167 | 32 | 153 | 32 | 163 | 88 | 41 | 0 |

Line Number IF INT (A) < 5 THEN PRINT TAB(X)
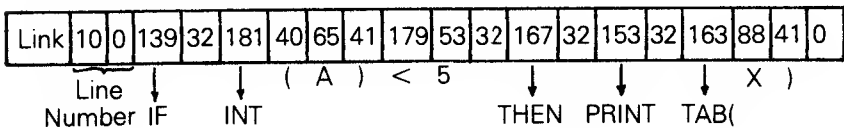
between commands which share the first two letters, as in the STOP and STEP. In these cases the first two letters should be typed followed by the third with the shift key depressed. Table 2 is a list of Basic commands and their abbreviated form with the numerical value of the command token in both decimal and hexadecimal.

The token value given to a Basic command is a pointer into a table of reserved command words located between 49310 and 49566. By subtracting 127 from the token value the number of the word in that table can be obtained. It should be noted that the technique of using tokens to represent words can give the programmer a very powerful method of generating print statements without consuming a large amount of memory. This can prove especially useful in games programs, such as Adventure, which require a lot of text generation. A table of, say, 200 common words is constructed and each time one of these words appears in a print statement it is represented by a number pointing to its location in the table. Obviously some sort of output subroutine is required to convert the token back into a word but the saving in memory space can be considerable, especially if done using machine code routines.

Having converted the Basic command into a single byte token the line is stored together with the line number and a link address at a location just above that of the last line entered. Assuming it is the first line of a program being entered on a standard VIC, then it will be entered into the following locations using the following format.

4096 — contents 0
4097 — link address low ⎱
                  ⎰ points to starting
                    location of next line
4098 — link address high ⎰
4099 — line number low
4100 — line number high
4101 — start of compressed Basic text.
        Number of bytes occupied variable.
        End of line flagged by a zero byte.

A Basic program is stored as a series of blocks each of variable length and representing one line in the program. Each block having a fixed format and all blocks being connected via a linked list structure. Each line in a program is stored in memory in the correct position dictated by the magnitude of its line number, thus it will be the line
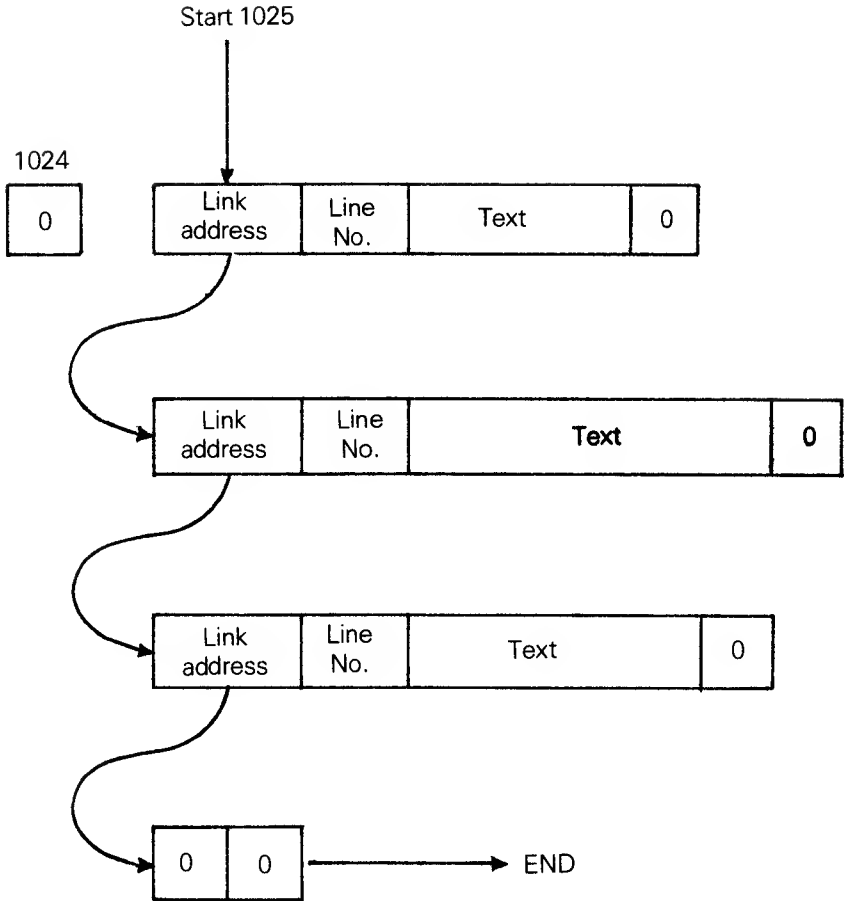
Fig. 9 — How a Basic program is stored in memory.

with the lowest number which is stored at the bottom of memory — location 4097 up. The line number is stored in byte 3 and 4 of a block in binary format. This means that the largest line number that can be used in a program is 65535, any number above that will give a syntax error. When a program is run the current line number being executed is stored in locations 57 and 58. A direct mode of operation for the processor is indicated when the contents of these two bytes is zero. The double byte link address points to the starting byte of the next line. As each line is executed this address is stored in locations 122 and 123, where it is accessed when the operating system fetches the next line. The link address of the last line of a program points not to another link address as in a normal program line, but to two bytes the contents of which are zero. The storage of a program within memory is best illustrated by the diagram in Figure 9.

A knowledge of how a program is stored in memory is useful, enabling several operations not otherwise allowed by the system to be performed; line renumbering, program margins and overlays. Line numbers can be changed simply by changing the contents of bytes three and four of each block (line). The beginning of each line is located using the link address obtained from the previous line. It should be noted however that this will not renumber any of the jump addresses stored in the Basic text. To do this the program must examine the tokens used in the Basic text area, looking for GOTO or GOSUB commands and renumber their jump addresses. Whereas the line number is stored in a binary format the jump line number is stored in ASCII and is thus of variable length.

DATA STORAGE

The entire area of memory not used for program storage is available for storage of data. Firstly, it is worth looking at the simplest form of data storage — using data statements. A data statement is stored as part of a program in the Basic text area of memory. The data is accessed by the program using the READ command. Data stored in data statements can only be added to by adding program lines. Another limitation is that data can only be accessed from data statements in a serial mode, meaning that to find one particular item the whole table of data must be read. The pointer to the current data statement is stored in locations 65 and 66 and the data line in 63 and 64. Manipulation of the contents of these locations could provide the user with a means of overcoming the serial search limitation.

Data not stored within the program as data statements, is stored by the program in the area of memory above the Basic text area, as variables. Variables can be divided into two groups. Simple variables of the kind used in the following statement; LET X = 47 where X is a simple variable. Array variables are defined by a DIM statement and contain more than one value. The number of values is determined by the number of elements in the DIM statement. For both groups of variables there are three types of data — real or floating point numbers — integer numbers — and character or string variables, (where words are being stored rather than numbers).

Simple variables of whatever data type are stored immediately above the Basic program text area, at an address pointed to by the contents of locations 45 and 46. The amount of memory used to store these variables depends on the number of variables used by a program. Each variable occupies seven bytes of memory and the next free location in the simple variable storage area is pointed to by the contents of locations 47 and 48.

The array variables are stored above the simple variables and thus start from the location pointed to by 47 and 48. The amount of memory used to store the array variables depends on the number of array variables, the number of elements in each and the data type of each variable. The end of the storage area used for array variables which is also the beginning of the unused storage area of memory, is pointed to by locations 49 and 50. Since array variables are stored directly above simple variables, whenever a new simple variable is encountered in a program, the operating system shifts the entire

65

INTEGER VARIABLES

first second character in variable name (the ASCII value + 128)	high low order byte of binary representation of integer value	0	0	0

FLOATING POINT VARIABLE

first second character in variable name	binary exponent + 129	binary mantissa in packed BCD giving eight digit precision. First bit of first byte is sign bit.

STRING VARIABLES

first second character in variable name, 128 added to ASCII value of second character only.	number of characters	low high order byte of address where string is stored	0	0

Fig. 10 — The storage of Basic variables in memory.

array variable storage area up seven bytes in memory thereby opening up a space to accommodate the new variable. This dynamic re-allocation of data storage space is one of the reasons why a machine code subroutine can not be stored in unused memory space, unless placed above the address stored in the top of memory pointers in locations 55 and 56. The re-allocation of memory space slows down a program, every time a new variable is encountered processing stops while the data is moved. When processing speed is important, such as in real time applications, this rather inconsistent variation in speed can be a problem. It is overcome by initialising all the variables — using dummy constants if necessary - at the beginning of the program.

Single value variables are divided into three distinct data types, each being stored in a different format. The only thing all three have in common is that each variable stored requires seven bytes of memory. Both integer and floating point numbers stored as single value variables have both the name and the value stored within the seven bytes allocated to each variable. An integer variable is distinguished from a floating point variable by adding 128 to the ASCII value of the variable name. The formats used are shown in Figure 10. From this, one can see that there is no saving in memory usage by using single value integer variable instead of floating point variables.

When the data being stored consists of a string of alphanumeric characters then the variable is stored using the character format. In this format the data is not stored within the seven bytes allocated for variable storage. What is stored is a pointer to an address in memory where this string of characters is stored. Character strings are in fact stored in an area right at the top of memory and extending downwards towards the area occupied by the array variables. By using this method string variables need not be of a fixed length thereby considerably reducing the amount of memory needed to store them. The format used for a string variable is shown in Figure 10.

Since the number of characters in the string is stored as a single byte it is not possible to have a character string longer that 255 characters. This should be considered when adding two string variables together where both are fairly long. Though the area at the top of memory is allocated for the storage of strings, not all string variables are stored there. Thus all strings defined within the program are retrieved, when required from the program text area. This is done

Array header	Element No. 0	Element No.1	Element No. 2		Element No. N

first second characters in array name, plus data type coding if any	low high pointer to first byte of next array	number of dimens- ions in array	high low number of elements in the last specified dimension of the array	expansion bytes
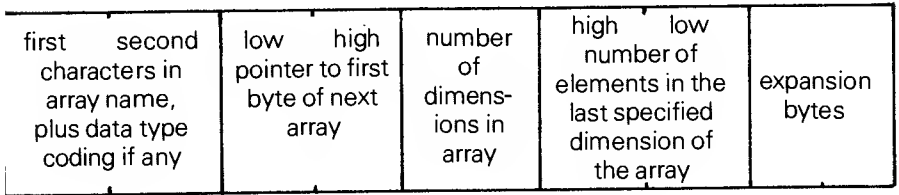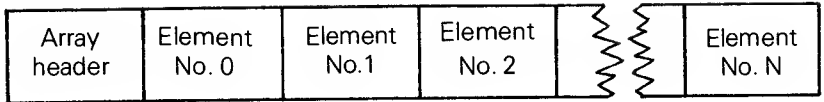
FLOATING POINT ARRAY ELEMENT

binary exponent plus 129	binary mantissa, first byte bit 7 is used to indicate the sign.

INTEGER ARRAY ELEMENT

high   low order byte of binary integer value
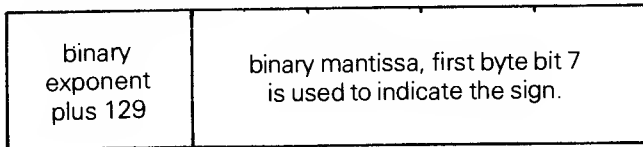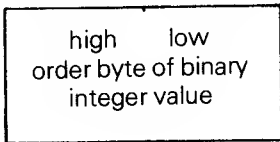
CHARACTER ARRAY ELEMENT

number of characters in string	low high byte of address where string is stored
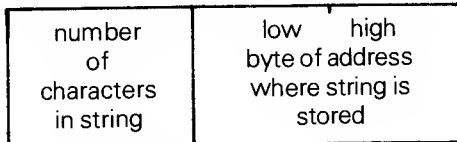
Fig. 11 — The storage of Basic array variables in memory.

by having the variable address pointers point to the location in Basic text rather than the top of memory. What is stored at the top of memory are calculated string variables. The area of memory occupied by these strings can be determined by looking at the contents of locations 51 and 52 this is the start address of the string area, and 53 and 54 which is the end address.

The three data types encountered as simple single value variables can also be stored as multiple value or array variables. Whereas simple variables of whatever data type all occupy the same amount of memory for each variable, the memory requirement for an array is different for each type of data. An array is stored as; an array header plus a set of elements each roughly corresponding to a simple variable. The array header contains the array name, the number of dimensions in the array, the number of elements in each dimension together with a pointer to the start of the next array. Array headers are the same for all data types. As with simple variables the array data type is coded into the array name. In a floating point array both characters are the normal ASCII code. In an integer array 128 is added to the ASCII value of both characters, and in a character array 128 is added to the ASCII value of the second character only. The general format of an array is shown in Figure 11. Here N is used to designate the last element in an array and corresponds to the value used in the DIM statement at the beginning of the program when the array was initialised. The array header for whatever data type has the format shown in Figure 11.

In a one dimensional array the array header occupies seven bytes, but if two dimensions are specified then an extra two bytes are required to specify the number of elements in that dimension, making the header nine bytes long. Similarly if there are three dimensions it would be eleven bytes long. In a two dimensional array set up by DIM D(A, B) the number of elements in B is stored in bytes 6 and 7 of the header, the number of elements in A is stored in bytes 8 and 9. The format for each element in an array is identical since all elements are of the same data type, though the format is different for each data type, these are shown in Figure 11.
NOTE: a negative integer whether in an array or a simple variable is stored as a two's complement number, thus a negative integer cannot exceed 32768.

Programs involving extensive string manipulation can suffer from seemingly inexplicable and often lengthy pauses in their operation. This is caused by an operating system function known as garbage collection. Every time a character string is input, or calculated, it is stored at the bottom of the character string storage area in a string. To avoid running out of memory the system must perform at this point a "garbage collection" routine. Garbage collection reclaims all the unused memory and compacts the string storage at the top of memory. This subroutine which is located at $D526 is lengthy and time consuming especially in large programs and is the main reason why such programs run at a much slower rate than small programs. One can force garbage collection to take place by performing the command FRE (U) which calculates the amount of free memory space. This is useful if you don't want a real time program interrupted by the garbage collection process. Generally the more user memory there is available in the system, coupled with extensive string manipulation in a program, the longer the delays caused by garbage collection.

THE BASIC AND OPERATING SYSTEM SOFTWARE

The top 16K of memory is occupied by the system software, these are the programs which allow the VIC to be programmed in Basic, to display and input data, and communicate with peripheral devices. This 16K of machine code programs is very important since it defines the VIC as a system, the VIC hardware is very flexible and by changing the system software the VIC could become a totally different machine. There is nothing to stop the user from completely re-defining the VIC interfaces to conform to say Centronics standard rather than RS232, it just requires a change in the operating system software. Similarly the VIC could be converted to run any high level language instead of Basic simply be replacing the Basic interpreter software. This flexability is an extremely valuable feature of the VIC since it allows the programmer to re-define the system to suit a particular application.

The 16K of system software can be divided into two distinct sections, the Basic interpreter and the operating system kernal. Each of these two sections are approximately the same length and each is contained on its own 8K ROM. The Basic interpreter ROM lies in memory space from address hex $0000 to $DFFF, the operating system kernal ROM lies from address hex $E000 to $FFFF. The operating system kernal is a totally self contained program and does not need the Basic interpreter program to function. The Basic interpreter however, uses the operating system routines to perform all I/O and peripheral communications functions. Both Basic and the operating system transfer variables between their constituent routines and between the two programs using the RAM space allocated to variables at the bottom of memory and processor registers are also used.

In Basic most of the calculations are performed using floating point numbers rather than simple integers or binary values. Consequently most of the routines which perform these functions utilise one or both of the floating point accumulators, both are located in page zero memory, they have the following format and location:

	Location	
Acc No. 1	Acc No. 2	Function
$61	$69	Exponent + $80
$62	$6A	Fraction MSB (binary)
$63	$6B	Fraction byte 2
$64	$6C	Fraction byte 3

71

$65	$6D	Fraction LSB
$66	$6E	Sign (FF = — and 0 = +)
$6F		Sign comparison byte
$70		Rounding byte for Acc No. 1

The majority of routines within both Basic and the operating system can be accessed and used by other machine code programs requiring that function, this can greatly reduce the amount of code required. To use these routines one needs to know the entry point and the nature, location and format of any parameters passed between the routine and the calling program. The designers of the VIC have made it fairly easy to use 36 of the most useful routines in the operating system kernal, by making them accessible through a jump table. Other routines in the kernal and Basic are less easy to use, particularly in Basic since this software originated outside Commodore (it was originally written by Microsoft but ammended by Commodore). All the major system software entry points are listed in Table 3 together with a short description of the function of each routine. The following is a description of the most useful of these routines, how their parameters are passed and how they can be used from a user written machine code program.

$C43A — Error Message Handling Routine

Communication registers: message No. is in X reg.

Description: This routine outputs an error message from the table of error messages, the message number is contained in the X index register. The error message is output to the currently open output device (default to screen). This is a useful way of generating error messages in a user program though one is limited to the standard set of messages.

$C483 — Main Command Handling Routine.

Description: This routine handles a new Basic line input from the keyboard and either executes it in the direct mode or stores it in indirect mode. This routine will be required by the programmer when adding extra commands to Basic.

$C560 — Input and Place in Basic Buffer.

Communication registers: 89 byte Basic buffer locations $0200 to $0258

NAME	FUNCTION
C000—C045	Action addresses for primary keywords
C046—C073	Action addresses for functions
C074—C091	Hierarchy and action addresses for operators
C092—C192	Table of Basic keywords
C193—C2A9	Basic messages, mostly error messages
C38A—C3B7	Search stack for FOR or GOSUB activity
C3B8—C3FA	Open up space in memory
C3FB—C407	Test: stack too deep?
C408—C434	Check available memory
C435	Send canned error message, then:
C474—C482	Print Ready
C483—C532	Handle new Basic line from keyboard
C533—C55F	Rebuild chaining of Basic lines in memory
C560—C57B	Receive line from keyboard
C57C—C612	Change keywords to Basic tokens
C613—C641	Search Basic for a given Basic line number
C642	Perform NEW, then:
C660—C68D	Perform CLR
C68E—C69B	Reset Basic execution to start-of-program
C69C—C741	Perform LIST
C742—C7EC	Perform FOR
C7ED—C81G	Execute Basic statement
C81D—C82B	Perform Restore
C82C—C856	Perform STOP and END
C857—C870	Perform CONT
C871—C882	Perform RUN
C883—C89F	Perform GOSUB
C8A0—C8D1	Perform GOTO
C8D2—C8EA	Perform RETURN, and perhaps:
C8EB—C905	Perform DATA, i.e., skip rest of statement
C906—C908	Scan for next Basic statement
C909—C927	Scan for nextBasic line
C928—C93A	Perform IF, and perhaps:
C93B—C94A	Perform REM, i.e., skip rest of line
C94B—C96A	Perform ON
C96B—C9A4	Get fixed-point number from Basic
C9A5—CA1C	Perform LET

CA1D—CA2B	Add ASCII digit to accumulator No. 1.
CA2C—CA7F	Continue to perform LET
CA80—CA85	Perform PRINT #
CA86—CA99	Perform CMD
CA9A—CB1D	Perform Print
CB1E—CB3A	Print string from memory
CB3B—CB4C	Print single format character (space, cursor-right,?)
CB4D—CB7A	Handle bad input data
CB7B—CBA4	Perform GET
CBA5—CBBE	Perform INPUT No.
CBBF—CBF8	Perform INPUT
CBF9—CC05	Prompt and receive input
CC06—CCFB	Perform READ; common routines used by INPUT and GET
CCFC—CD1D	Messages: EXTRA IGNORED, REDO FROM START
CD1E—CD77	Perform NEXT
CD78—CD9D	Check data type, print TYPE MISMATCH
CD9E—CEF0	Input & evaluate any expression (numeric or string)
CEF1—CEF6	Evaluate expression within parentheses ( )
CEF7—CEF9	Check right parenthesis)
CEFA—CEFC	Check left parenthesis (
CEFD—CF07	Check for comma
CF08—CF0C	Print SYNTAX ERROR and exit
CF0D—CF13	Set up function for future evaluation
CF14—CFA6	Search for variable name
CFA7—CFE5	Identify and set up function references
CFE6—CFE8	Perform OR
CFE9—D015	Perform AND
D016—D07D	Perform comparisons, string or numeric
D07E—D08A	Perform DIM
D08B—D112	Search for variable location in memory
D113—D11C	Check if ASCII character is alphabetic
D11D—D193	Create new Basic variable
D194—D1A4	Array pointer subroutine
D1A5—D1A9	32768 in floating binary
D1AA—D1D0	Evaluate expression for positive integer
D1D1—D34B	Find or create array
D34C—D37C	Compute array subscript size
D37D—D390	Perform FRE then:
D391—D39D	Convert fixed point to floating point
D39E—D3A5	Perform POS
D3A6—D3B2	Check if direct command, print ILLEGAL DIRECT

D3B3—D3E0	Perform DEF
D3E1—D3F3	Check FNx syntax
D3F4—D464	Evaluate FNx
D465—D474	Perform STR$
D475—D486	Calculate string vector
D487—D4F3	Scan and set up string
D4F4—D525	Subroutine to build string vector
D526—D5BC	Garbage collection subroutine
D5BD—D605	Check for most eligible string collection
D606—D63C	Collect a string
D63D—D679	Perform string concatenation
D67A—D6A2	Build string into memory
D6A3—D6DA	Discard unwanted string
D6DB—D6EB	Clean the descriptor stack
D6EC—D6FF	Perform CHR$
D700—D72B	Perform LEFT$
D72C—D72C	Perform RIGHT$
D737—D760	Perform MID$
D761—D77B	Pull string function parameters from stack
D77C—D781	Perform LEN
D782—D78A	Move from string-mode to numeric-mode
D78B—D79A	Perform ASC
D79B—D7AC	Input byte parameter
D7AD—D7EA	Perform VAL
D7EB—D7F6	Get two parameters for POKE or WAIT
D7F7—D80C	Convert floating point to fixed point
D80D—D823	Perform PEEK
D824—D82C	Perform POKE
D82D—D848	Perform WAIT
D849—D84F	Add 0.5 to accumulator No. 1.
D850—D861	Perform subtraction
D862—D946	Perform addition
D947—D97D	Complement accumulator No. 1
D97E—D982	Print OVERFLOW and exit
D983—D9BB	Multiply-a-byte subroutine
D9BC—D9E9	Function constants: 1, SOR(.5), SOR(2), –00.5. etc.
D9EA—DA2F	Perform LOG
DA30—DA58	Perform multiplication
DA59—DA8B	Multiply-a-bit subroutine
DA8C—DAB6	Load accumulator No. 2 from memory
DAB7—DAD3	Test and adjust accumulators No. 1 and No. 2.
DAD4—DAE1	Handle overflow and underflow

DAE2—DAF8	Multiply by 10
DAF9—DAFD	10 in floating binary
DAFE—DB06	Divide by 10
DB07—DB11	Perform divide-into
DB12—DBA1	Perform divide-by
DBA2—DBC6	Load accumulator No. 1 from memory
DBC7—DBFB	Store accumulator No. 1 into memory
DBFC—DC0B	Copy accumulator No. 2 into accumulator No. 1.
DC0C—DC1A	Copy accumulator No. 1 into accumulator No. 2.
DC1B—DC2A	Round off accumulator No. 1.
DC2B—DC38	Compute SGN value of accumulator No. 1.
DC39—DC57	Perform SGN
DC58—DC5A	Perform ABS
DC5B—DC9A	Compare accumulator No. 1 to memory
DC9B—DCCB	Convert floating-point to-fixed-point
DCCC—DCF2	Perform INT
DCF3—DD7D	Convert string to floating-point
DD7E—DDB2	Get new ASCII digit
DDB3—DDC1	String conversion constants: 99999999,999999999
	1E+9
DDC2	Print IN, followed by:
DDCD—DDDC	Print Basic line number
DDDD—DF10	Convert number or TI$ to ASCII
DF11—DF70	Constants for numeric conversion
DF71—DF77	Perform SQR
DF78—DFB3	Perform power function
DFB4—DFBE	Perform negation
DFBF—DFEC	Constants for string evaluation
DFED—E03F	Perform EXP
E040—E089	Function series evaluation subroutines
E08A—E093	Manipulation constants for RND
E094—E0F5	Perform RND
E0F6—E260	Kernal patch routines (see Appendix 6 for listings)
E261—E267	Perform COS
E268—E2B0	Perform SIN
E2B1—E2DC	Perform TAN
E2DD—E30A	Constants for trig evaluation pi/2, 2No.pi, .25, etc.
E30B—E33A	Perform ATN
E33B—E377	Constants for ATN series evaluation
E378—E386	Initialise RAM vectors
E387—E3A3	Subroutine to be moved to zero page ($70 to $87)
E3A4—E428	Initialise Basic system

E429—E44E	Messages: BYTES FREE, **** CBM BASIC V2 ****
E44F—E47B	Vector initialisation (see Appendix 6 for listings)
E47C—E4FF	Unused space

KERNAL ROUTINES

E500—E504	Return address of 6522
E505—E509	Return max rows and columns of screen
E50A—E517	Read/plot cursor position
E518—E580	Initialise I/O
E581—E586	Home function
E587—E5B4	Move cursor to current line index pointer
E5B5—E5C2	Panic NMI entry (Restore key)
E5C3—E5CE	Initialise 6561 VIC chip
E5CF—E64E	Remove character from queue
E64F—E741	Input a line until carriage return
E742—E8E7	Print routine
E8E8—E8F9	Check for decrement in line index pointer
E8FA—E911	Check for increment in line index pointer
E912—E928	Check colour
E929—E974	Table to convert from screen code to ASCII
E975—EAA0	Screen scroll routines
EAA1—EB1D	IRQ routines, put char on screen and update time, generate I/O
EBIE—EC45	General keyboard scan
EC46—EE13	Keyboard matrix tables
EE14—EEBF	Command serial bus device to listen
EEC0—EEC4	Send secondary address after listen
EEC5—EECD	Release attention after listen
EECE—EEE3	Talk second address
EEE4—EEF5	Buffered output to serial bus
EEF6—EF03	Send untalk command on serial bus
EF04—EF18	Send unlisten command on serial bus
EF19—EFA2	Input a byte from serial bus
EFA3—EFED	NMI continue routine
EFEE—F035	Transmit byte
F036—F173	NMI routine to collect data into bytes (RS-232)
F174—F1E1	Kernal messages
F1E2—F1F4	Print message to screen
F1F5—F20D	Get character from channel
F20E—F279	Input character from channel

F27A—F2C6	Output character to channel
F2C7—F308	Open channel for input
F309—F349	Open channel for output
F34A—F3EE	Close logical file
F3EF—F3F2	Close all logical files
F3F3—F409	Clear channels
F40A—F541	Open function
F542—F674	Load RAM function (from cassette or bus devices)
F675—F733	Save function
F734—F76F	Time function
F770—F77D	Test stop key
F77E—F7AE	Error handler
F7AF—F889	Find and read tape header
F88A—F98D	Cassette control routines
F98E—FABC	Tape read routines
FABD—FBE9	Byte handler for cassette read
FBEA—FD21	Tape write routines
FD22—FE90	System power up initialisation
FE91—FEA8	Memory check routines
FEA9—FF5B	NMI handler
FF5C—FF71	Baud rate tables
FF72—FF85	IRQ handler
FF85—FFFF	Kernal jump vector addresses

Description: Data strings up to 88 characters long are input by this routine and stored in the Basic input buffer. The buffer is filled starting at location $0200 upwards, end of string terminated by a zero byte.

$C57C — Tokenise Basic Command.

Description: Basic commands are converted to single byte tokens by this routine, reducing memory requirements for program storage. Routine required when adding commands to Basic.

$CBIE — Print String Pointed to by Y, A

Communication registers: Y index and Accumulator.

Description: A data string is printed on the current output device, default device is the screen. The memory address of the start of the string is pointed to by the contents of the Y index register (LSB of address) and the Accumulator (MSB of address). The end of the string is the first byte encountered containing a binary zero.

$CE86 — Evaluate Expression.

Communication registers: Page Zero $7A and $7B plus Stack and Accs No. 1 and No. 2

Description: This routine evaluates a Basic expression starting at an address stored in locations $7A (LSB of address and $7B (MSB of address). The result is stored in Accumulator No. 1.

$CFE6 — Logical OR between contents of Acc No. 1 and Acc No. 2.

Communication registers: Floating point Accumulators No. 1 and No. 2.

Description: A logical OR is performed between values contained in the two floating point accumulators, the result is placed in accumulator No. 1.

$CFEB — Logical AND between contents of Acc No. 1 and Acc No. 2.

Communication registers: Floating point Accumulators No. 1 and No. 2.

Description: A logical AND is performed between values contained in the two floating point accumulators, the result is placed in accumulator No. 1.

$D1AA — Convert Floating Point Number to Integer.

Communication registers: Floating point Accumulator No. 1.

Description: A number in floating point format stored in Accumulator No. 1 is converted by this routine to a double byte integer stored in two bytes of Accumulator No. 1. The two bytes used are $64 and $65, the format of the integer number is 100* $64 + $65.

$D37D — Perform FRE function.

Communication registers: Floating Accumulator No. 1.

Description: This function determines the number of free bytes of memory available in the system for user program or data storage. The arguments of the function are stored and returned as a floating point number in Accumulator No. 1.

$D391 — Integer to Floating Point conversion.

Communication registers: Y index register, Accumulator, and Floating Accumulator No. 1.

Description: A two byte integer value stored in Y index register and Accumulator is converted to a floating point number stored in floating accumulator No. 1. The integer value is stored in the format — 100 * accumulator + Y index register.

$D77C — Perform LEN function.

Communication registers: X index register, and Floating accumulator No. 1.

Description: This routine calculates the number of characters in a string, the argument of the function, ie. the string name, is stored in bytes $64 and $65 of floating accumulator No. 1. The string length is returned in the X index register.

```
┌─────────────────────────────┐
─  LOOP  RETURN  ADDRESS
├─────────────────────────────┤
─  RETURN  LINE No
├─────────────────────────────┤
─
─
─        "TO"  VALUE
─
─
├─────────────────────────────┤
        SIGN  OF  STEP
├─────────────────────────────┤
─
─
─        STEP  VALUE
─
─
├─────────────────────────────┤
─  VARIABLE'S  ADDRESS
├─────────────────────────────┤
              $81
└─────────────────────────────┘
```

FOR – NEXT

```
┌─────────────────────────────┐
─  $ F6FC
├─────────────────────────────┤
─  RETURN  ADDRESS
├─────────────────────────────┤
─  RETURN  LINE  No
├─────────────────────────────┤
              $ 8D
└─────────────────────────────┘
```
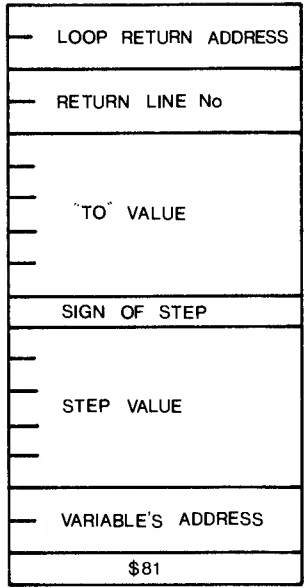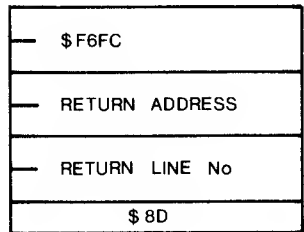
GO – SUB

Fig. 12 — Stack usage by two Basic interpreter routines.

$D850 — Subtract Acc No. 2. from Acc No. 1.

Communication registers: Floating Accumulators No. 1 and No. 2. and Accumulator.

Description: The contents of floating point accumulator No. 2. is subtracted from the contents of floating accumulator No. 1. by this routine, the result being stored in accumulator No. 1. Before this routine is called the sign comparison byte $6F must be set, this is done by exclusively ORing the contents of $66 and $6E and storing the result in $6F, the processor accumulator should also contain the value stored in location $61 (MSB of Acc No. 1.).

$D9EA — Perform LOG function.

Communicating registers: Floating point accumulator No. 1.

Description: This routine performs the LOG function, the value used in the functions argument is stored in floating accumulator No. 1. the result is placed in the same accumulator.

$DA30 — Multiply Floating Point Number in Memory by Acc No. 1.

Communicating registers: Floating point accumulators No. 1 and No. 2., accumulator and Y index register.

Description: This routine first obtains the contents of floating accumulator No. 2 from memory. The memory location is a two byte address stored in the processor accumulator and Y index register, the format used is 100 * Y index + accumulator. The value stored in accumulator No. 2. is then multiplied by the contents of accumulator No. 1 and the result stored in accumulator No. 1.

$DA33 — Multiply Acc No. 2. by Acc No. 1.

Communicating registers: Floating point accumulators No. 1. and No. 2. and processor accumulator.

Description: The contents of floating point accumulator No. 2. is multiplied by the contents of accumulator No. 1., and the result stored in accumulator No. 1. Before using this routine the sign comparison in $6F should be set, this is done by exclusively ORing the contents of the two sign bytes $66 and $6E and storing the result in $6F. The

exponent of the value in accumulator No. 1., stored in $61, should be loaded into the processor accumulator prior to running this routine.

$DA8C — Move Contents of Memory to Acc No. 2.

Communicating registers: Floating point accumulator No. 2., processor accumulator and Y index register.

Description: This routine takes a value stored in memory and loads it into floating point accumulator No. 2. The two byte memory address for the value is stored in the processor accumulator and Y index register, the format used is accumulator + 100 * Y index register. The routine seperates the sign byte and sets the sign comparison byte, the contents of $61, the exponents of accumulator No. 1 are loaded into the processor accumulator.

$DAE2 — Multiply Accumulator No. 1 by 10.

Communicating registers: Floating Point Accumulators No. 1 and No. 2.

Description: The contents of floating point accumulator No. 1 is multiplied by 10 and the result is stored in floating point accumulator No. 2.

$DAFE — Divide Accumulator No. 1. by 10.

Communicating registers: Floating Point Accumulators No. 1 and No. 2.

Description: The contents of floating point accumulator No. 1 is divided by 10 and the result is stored in floating point accumulator No. 2.

$DBOF — Divide accumulator No. 2. by Accumulator No. 1.

Communicating registers: Floating Point Accumulators No. 1 and No. 2.

Description: This routine divides the contents of accumulator No. 2. by the contents of accumulator No. 1. and puts the result in accumulator No. 1. Before running this routine the sign comparison in

83

$6F should be set, this is done by exclusively ORing the contents of the two sign bytes $66 and $6E and storing the result in $6F. The exponent of the value in accumulator No. 1., stored in $61, should be loaded into the processor accumulator prior to running this routine.

$DBA2 — Move Contents of Memory to Acc No. 1.

Communicating registers: Floating Point Accumulator No. 1., processor accumulator and Y index register.

Description: This routine takes a value stored in memory and loads it into floating point accumulator No. 1. The two byte memory address for the value is stored in the processor accumulator and Y index register, the format used is accumulator + 100 * Y index register. The routine seperates the sign byte and sets the sign comparison byte, the contents of $61, the exponents of accumulator No. 1, is loaded into the processor accumulator.

$DBC7 — Move Contents of Accumulator No. 1. to Memory.

Communicating registers: Floating Point accumulator No. 1. X and Y index registers.

Description: The value in floating point accumulator No. 1. is stored in a specified memory location by this routine. The two byte memory address is stored in the X and Y processor index registers, the format used is X index + 100 * Y index register. The routine merges the sign byte to give the correct memory storage format (the first bit of first byte = sign).

$DBFC — Transfer Contents of Acc No. 2. to Acc No. 1.

Communicating registers: Floating Point Accumulators No. 1. and No. 2.

Description: The current contents of floating point accumulator No. 2., are copied into accumulator No. 1., the contents of accumulator No. 2 remain unchanged.

$DCOC — Transfer Contents of Acc No. 1., to Acc No. 2. with Rounding.

Communicating registers: Floating Point Accumulators No. 1., and No. 2.

Description: The contents of floating point accumulator No. 1., are copied into accumulator No. 2., the contents of accumulator No. 1., are then rounded and if necessary the exponent adjusted.

$DCOF — Transfer Contents of Acc No. 1., to Acc No. 2.

Communicating registers: Floating Point accumulators No. 1., and No. 2.

Description: The contents of floating point accumulator No. 1., are copied into accumulator No. 2., the contents of accumulator No. 1., are then rounded and if necessary the exponent adjusted.

$DC58 — Performs ABS function.

Communicating registers: Floating Point Accumulator No. 1.

Description: The absolute value of the contents of floating point accumulator No. 1. are returned to accumulator No. 1 by this routine.

$DC39 — Perform SGN function.

Communicating registers: Floating Point Accumulator No. 1.

Description: This routine returns the sign of a value stored in floating point accumulator No. 1. If the value in accumulator No. 1., is greater than 0 then a 1 is stored in accumulator No. 1., if it equals zero then a 0, and if less than zero then a —1.

$DC5B — Compare Contents of Acc No. 1., to Memory.

Communicating registers: Floating Point Accumulator No. 1., processor accumulator and Y index register.

Description: The current contents of floating point accumulator No. 1., are compared to a floating point variable stored in memory, and the processor accumulator set to a value dependant on whether the two variables are equal or not. The two byte address for the floating point variable in memory is stored in the processor accumulator and the Y index register, the format used is accumulator + 100 * Y index. If the two floating point variables are equal then the processor accumulator is set to $00, and if not equal then it is set to $FF.

$DC9B — Convert Floating Point Variable to Fixed Point.

Communicating registers: Floating Point Accumulator No. 1.

Description: A floating point variable stored in accumulator No. 1., is converted to a fixed point format by this routine, the fixed point value is stored in accumulator No. 1.

$DCCC — Perform INT function.

Communicating registers: Floating Point Accumulator No. 1.

Description: This routine converts a floating point variable stored in accumulator No. 1., into an integer value, the result is stored back in accumulator No. 1.

$DF71 — Perform SQR function.

Communicating registers: Floating Point Accumulator No. 1.

Description: The square root of a floating point variable stored in accumulator No. 1., is calculated by this routine, the result also in floating point format is returned in accumulator No. 1.

$DF78 — Raise Acc No. 2., to the power of Acc No. 1.

Communicating registers: Floating Point Accumulators No. 1., and No. 2.

Description: The contents of floating point accumulator No. 2., is raised to the power of a value stored in accumulator No. 1., the result is placed in accumulator No. 1. Before using this routine the sign

comparison in $6F should be set, this is done by exclusively ORing the contents of the two sign bytes $66 and $6E and storing the result in $6F. The exponent of the value in accumulator No. 1., stored in $61, should be loaded into the processor accumulator prior to running this routine.

$DFED — Perform the EXP function.

Communicating registers: Floating Point Accumulator No. 1.

Description: This routine calculates 'e' to the power of the value in floating point accumulator No. 1., and places the result in accumulator No. 1.

$E094 — Perform the RND function.

Communicating registers: Floating Point Accumulator No. 1, plus page zero locations $8B to $90.

Description: A random value is created by this routine and placed in floating point accumulator No. 1. Prior to running the routine floating point accumulator No. 1., contains a seed value used to initialise the random number calculation routine, also memory locations $8B to $90 contain the last random number generated.

$E261 — Perform COS function.

Communicating registers: Floating Point Accumulator No. 1.

Description: The COSine of a value, in radians, stored in floating point accumulator No. 1., is calculated by this routine and the result placed in accumulator No. 1.

$E268 — Perform SIN function.

Communicating registers: Floating Point Accumulator No. 1.

Description: This routine calculates the SINe of a value, in radians, stored in floating point accumulator No. 1., the result is placed in accumulator No. 1.

$E2B1 — Perform TAN function.

87

Communicating registers: Floating Point Accumulator No. 1.

Description: This routine calculates the TAN of a value, in radians, stored in floating point accumulator No. 1., the result is placed in accumulator No. 1.

$E30B — Perform ATN function.

Communicating registers: Floating Point Accumulator No. 1.

Description: This routine calculates the ATN of a value, in radians, stored in floating point accumulator No. 1., the result is placed in accumulator No. 1.

$E378 — Initialise System Vectors and Variables.

Communicating registers: none.

Description: All system vectors and variables are initialised by this routine, it can be used together with its constituent subroutines to reinitialise system variables and vectors prior to returning to a Basic program from machine code.

USER CALLABLE KERNAL ROUTINES

NAME	ADDRESS		FUNCTION
	HEX	DECIMAL	
ACPTR	$FFA5	65445	Input byte from serial port
CHKIN	$FFC6	65478	Open channel for input
CHKOUT	$FFC9	65481	Open channel for output
CHRIN	$FFCF	65487	Input character from channel
CHROUT	$FFD2	65490	Output character to channel
CIOUT	$FFA8	65448	Output byte to serial port
CLALL	$FFE7	65511	Close all channels and files
CLOSE	$FFC3	65475	Close a specified logical file
CLRCHN	$FFCC	65484	Close input and output channels
GETIN	$FFE8	65512	Get character from keyboard queue (keyboard buffer)
IOBASE	$FFF3	65523	Returns base address of I/O devices
LISTEN	$FFB1	65457	Command devices on the serial bus to LISTEN
LOAD	$FFD5	65493	Load RAM from a device
MEMBOT	$FF9C	65436	Read/set the bottom of memory
MEMTOP	$FF99	65433	Read/set the top of memory
OPEN	$FFC0	65472	Open a logical file
PLOT	$FFF0	65520	Read/set X, Y cursor position
RDTIM	$FFDE	65502	Read real time clock
READST	$FFB7	65463	Read I/O status word
RESTOR	$FF87	65415	Restore default I/O vectors
SAVE	$FFD8	65496	Save RAM to device
SCNKEY	$FF9F	65439	Scan keyboard
SCREEN	$FFED	65517	Return X, Y organisation of screen
SECOND	$FF93	65427	Send secondary address after LISTEN
SETLFS	$FFBA	65466	Set logical, first, and second addresses
SETMSG	$FF90	65424	Control KERNAL messages
SETNAM	$FFBD	65469	Set file name
SETTIM	$FFDB	65499	Set real time clock
SETTMO	$FFA2	65442	Set timeout on serial bus
STOP	$FFE1	65505	Scan stop key
TALK	$FFB4	65430	Command serial bus device to TALK
TKSA	$FF96	65430	Send secondary address after TALK
UDTIM	$FFEA	65514	Increment real time clock
UNTLK	$FFAB	65451	Command serial bus to UNTALK
VECTOR	$FF84	65412	Read/set vectored I/O

USER CALLABLE KERNAL ROUTINES

The VIC operating system software has been specially designed to allow the easy access of subroutines within it. These subroutines can be used by a machine code routine calling either a ROM or RAM based vector address. The required variables having been previously passed to the subroutine via the processor registers. The main block of kernal vector addresses are stored at the top of ROM memory, a list is shown in Table 4. The smaller number of RAM vector addresses are stored in page three of memory and a list of these is shown in Table 1. The reason why some vector jump addresses are stored in RAM is that they can be changed. By changing the vector addressed, routines controlling the system I/0 and interrupt handling can be reconfigured. It should be noted that all the RAM vectors, except the interrupt handlers, point to routines which are also pointed to be ROM vectors. The following is a detailed description of each of the vector subroutines together with their function and use.

$FF8A — Restore Old I/0 Vectors.

Communicating registers: none.
Error Returns: none.
Stack Requirements: 2.
Preparatory routines: none.

Description: Restore default vector values for system subroutines and interrupts.

$FF8D — Read and Set Vectored I/0.

Communicating registers: X and Y index registers.
Error Returns: none.
Stack Requirements: 2.
Preparatory routines: none.

Description: If this routine is called with the carry bit set, it will then read the current contents of the RAM vectors and put them in a list starting at a memory location pointed to by (X, Y). When this routine is called with carry clear, the user list pointed at by (X, Y) is transferred to the system RAM vectors. When using this routine the best practice is to read first the entire contents of the vector table into a user memory area, alter the desired vectors, then empy the contents back into the system.

$FF90 — Control Kernal Messages.

Communicating registers: processor accumulator.
Error Returns: none.
Stack Requirements: 2.
Preparatory routines: none.

Description: This routine controls the printing of error and diagnostic messages by the kernal. It is called by placing a value in the accumulator. Bits 6 and 7 of this value control the message printing, bit 7 is set for kernal messages, and bit 6 for control messages. Bits 0 to 5 designate the message, and point to an entry in the error message tables.

$FF93 — Transmit Secondary Command.

Communicating registers: processor accumulator.
Error Returns: see routine $FFB7.
Stack Requirements:
Preparatory routines: $FFB1

Description: Sends a secondary address after 'listen' routine $FFB1. This routine cannot be used to send a secondary address after a 'talk' command from routine $FFB4.

$FF96 — Transmit Secondary After 'Talk'.

Communicating registers: processor accumulator.
Error Returns: see routine $FFB7.
Stack Requirements:
Preparatory routines: $FFB4

Description: Sends a secondary address for 'talk'. By loading the accumulator with a number between 0 and 31, the user sends a secondary address command over the IEEE with this subroutine. This routine can only be used after $FFB4, it will not work after $FFB1.

$FF99 — Read/Set Top of Memory.

Communicating registers: X and Y index registers.
Error Returns: none.
Stack Requirements: 2.
Preparatory routines: none.

Description: When this routine is called with carry set, the pointer

to the top of RAM is read into .X and .Y. A call with carry clear will copy the contents of .X and .Y into this pointer.

$FF9C — Read/Set Bottom of Memory.

Communicating registers: X and Y index registers.
Error Returns: none.
Stack requirements: 2.
Preparatory routines: none.

Description: A call to this routine with the carry bit set, causes the pointer to the bottom of RAM to be read into .X and .Y. The initial value is always $400. If the routine is called with carry clear then the contents of .X and .Y are transfered to the bottom of memory pointers.

$FF9F — Scan Keyboard.

Communicating registers: none.
Error Returns: none.
Stack Requirements:
Preparatory routines: none.

Description: This routine scans the keyboard, if a key is down, its corresponding ASCII code value is placed in the keyboard queue ($0277 to $0280). This is the same routine called by the interrupt handling routines every 1/60 second.

$FFA2 — Set Timeout on IEEE.

Communicating registers: processor accumulator.
Error Returns: none.
Stack Requirements: 2.
Preparatory routines: none.

Description: When the processor accumulator contains a 0 in bit 7, timeouts are enabled by this routine. A 1 in bit 7 disables timeouts. Timeouts are a method by which the VIC can poll an IEEE device for data without hanging in a timeshake sequence. The device must respond to DAV within 64 milliseconds. The VIC and CBM disks use the timeout to communicate a 'file not found' status in the OPEN command.

$FFA5 — Input byte from IEEE Bus.

Communicating registers: processor accumulator.
Error Returns: see routine $FFB7.
Stack requirements: 13.
Preparatory routines: $FFB4 and $FF96.

Description: This routine handshakes a byte off the IEEE bus. The data is returned in the processor accumulator. It is assumed that the device has been told to 'talk' by routine $FFB4 and it is possible that a secondary address has been sent by the routine $FF96.

$FFA8 — Output Byte to IEEE Bus.

Communicating registers: processor accumulator.
Error Returns: see routine $FFB7.
Stack Requirements:
Preparatory routines: $FFB1 and $FF93.

Description: The accumulator is loaded with a byte of data to handshake onto the IEEE bus. A device must be listening or status will show a timeout error (see routine $FFA2). One character is always buffered by this routine. When an 'unlisten' command is sent (by routine $FFAE), the buffered character is sent with the EOI line asserted, the 'unlisten' command is then sent.

$FFAB — Command IEEE Bus to 'Untalk'.

Communicating registers: none.
Error Returns: none.
Stack Requirements:
Preparatory routines: none.

Description: This sends an 'untalk' command to an IEEE device via the IEEE bus.

$FFAE — Command IEEE Bus to 'Unlisten'.

Communicating registers: none.
Error returns: none.
Stack Requirements:
Preparatoray routines: none.

Description: This sends an 'unlisten' command to an IEEE device via the IEEE bus.

93

$FFB1 — Command IEEE Device to 'Listen'.

Communicating registers: processor accumulator.
Error Returns: see routine $FFB7.
Stack Requirements:
Preparatory routines: $FFB1

Description: The IEEE command 'listen with attention' is performed by this routine. The processor accumulator is loaded with a device number between 0 and 30. This subroutine then ORs in bits to convert the device number to a 'listen' address and then transmits this data as a command on the IEEE bus.

$FFB4 — Command IEEE Device to 'Talk'.

Communicating registers: processor accumulator.
Error Returns: see routine $FFB7.
Stack Requirements:
Preparatory routines: none.

Description: The IEEE command 'talk with attention' is performed by this routine. The processor accumulator is loaded with a device number betwen 0 and 30. This subroutine ORs in bits to convert the device number into a 'talk' address and then transmits this data as a command on the IEEE bus.

$FFB7 — Read I/O Status Word.

Communicating registers: processor accumulator.
Error Returns: none.
Stack Requirements: 2
Preparatory routines: none.

Description: Returns the current I/O status. Usually checked after initiating any new communication to a channel. The bits in the byte returned contain the following data:

ST Bit Position	ST Numeric Value	Cassette Read	IEEE/RW	Tape Verify + Load
0	1		Time out/write	
1	2		Time out/read	
2	4	Short block		Short block
3	8	Long block		Long block
4	16	Unrecoverable		Any mismatch

		read error		
5	32	Checksum error		Checksum error
6	64	End of file	EOI line	
7	128	End of tape present	Device not present	End of tape

$FFBA — Set Logical, First, and Second Address

Communicating registers: processor accumulator, X and Y Index registers:.
Error Returns: none.
Stack Requirements: 2
Preparatory routines: none.

Description: Setting logical file number, device address, and command. The logical file number is used as a key by the system to access data stored in a table by the open file subroutine. The device address ranges from 0 to 30 and corresponds to the following VIC or CBM devices:

0 Keyboard
1 Cassette No. 1.
2 Cassette No. 2. (unused on VIC)
3 CRT display
4 IEEE printer
8 VIC or CBM IEEE disk drive

Device numbers 4 or greater correspond to devices on the IEEE bus.

Load the accumulator with the logical file number, X index register with the device number, and the Y index register with the command. The command is sent as a secondary address on the IEEE following the device number during an attention sequence. IF the programmer desires no secondary address to be sent, load Y index with a 255.

$FFBD — Set File Name Information.

Communicating registers: processor accumulator, X and Y index registers.
Error Returns: none.
Stack Requirements:
Preparatory routines: none.

Description: Should a file be opened without a file name, the name length must be set to zero. Load the accumulator with the length, X index with the low order address of the file name and Y with the high

order address. The file name address can be any valid memory address where the string of characters corresponding to the file name are stored.

$FFC0 — Open Logical File.

Communicating registers: none.
Error Returns: 1, 2, 4, 5 and 6.
Stack Requirements:
Preparatory routines: $FFBA and $FFBD.

Description: Open logical file to device. There are no arguments to be set up for this routine. Both $FFBA (Set logical number, device address and command) and $FFBD (Set file name information) must be called before calling this routine.

$FFC3 — Close Logical File.

Communicating registers: processor accumulator.
Error returns: none.
Stack Requirements:
Preparatory routines: none.

Description: Close a logical file to a device. When all I/0 to a file is completed this subroutine is called with the accumulator loaded with the logical file number used in the 'open' subroutine $FFC0.

$FFC6 — Open Channel for Input.

Communicating registers: X index register.
Error Returns: 3, 5 and 6.
Stack Requirements:
Preparatory routines: $FFC0

Description: Assuming that a file has been opened by subroutine $FFC0 (open logical file), it can be opened as an input channel. Of course the characteristics of the device will determine if it is valid to

do so. The logical file number if put in the X index register. This subroutine must be executed before subroutines $FFCF (input character from channel) or $FFE4 (get character from keyboard queue) are executed for a device other than the keyboard. If input from the keyboard is desired, and there is no association to the logical file number by a previous open file, then the call to this subroutine may be dispensed with. On the IEEE this subroutine results in sending a talk address followed by a secondary address if one was specified in the open subroutine ($FFC0).

$FFC9 — Open Channel for Output.

Communicating registers: X index register.
Error Returns: 3, 5 and 7.
Stack Requirements:
Preparatory routines: $FFC0.

Description: Assuming that a file has been opened by subroutine $FFC0 (open logical file), it can be opened as an output channel. Of course, the characteristics of the device will determine if it is valid to do so. This subroutine must be executed before subroutine $FFD2 (output character to channel) is executed for a device other than the CRT. If output to the CRT is desired, and there is no association to an open file by logical file number, then the call to this subroutine may be dispensed with. On the IEEE this subroutine results in sending a listen address followed by a secondary address if one was specified in the open subroutine ($FFC0).

$FFCC — Close Input and Output Channels.

Communicating registers: none.
Error Returns: none
Stack Requirements:
Preparatory routines: none.

Decription: After opening a channel and performing I/O, this routine closes all open channels and restores the default channels. Default input is device 0 (keyboard) and output device 3 (CRT screen). This routine may be called optionally by the programmer. An 'untalk' is sent to clear the input channel if the device is on the IEE. An 'unlisten' is sent to clear the output channel. By not calling this routine and leaving a listener addressed on the IEEE, multiple devices can receive data on the bus. An example would be to address the printer to listen and the disk to talk.

$FFCF — Input Character from Channel.

Communicating registers: processor accumulator.
Error Returns: see routine $FFB7.
Stack Requirements:
Preparatory routines: none.

Description: A call to this routine will return a character of data from the channel set up by a call to subroutine $FFC6 (open channel for input), or the default input channel if no other has been set up. Data is returned in the accumulator. The channel remains open after the call. In the case of the keyboard device, the cursor is turned on and continues to blink until carriage return is typed. Characters on the line are then returned one by one, by calls to this routine. Finally carriage return is sent and the process begins again.

$FFD2 — Output Character to Channel.

Communicating registers: processor accumulator.
Error Returns: see routine $FFB7.
Stack Requirements:
Preparatory routines: none.

Description: The data to be output is loaded into the accumulator. A call to $FFC9 (open channel for output) sets up the output channel, or if this call is omitted, data is sent to the default device which is number 3, the CRT. The character can be transmitted to multiple devices on the IEEE if a clear channel is not performed after the corresponding open channel for output.

$FFD5 — Load RAM from Device.

Communicating registers: processor accumulator, X and Y index registers.
Error Returns: 0, 4, 5, 8 and 9.
Stack requirements:
Preparatory routines: $FFBA and $FFBD.

Description: Load from device into RAM. On call accumulator = 0 for load, or accumulator = 1 for verify. The index registers (X, Y) contain the address to load into for secondary address = 3. If the secondary address = 0, 1 or 2 then the block will load into memory starting at the address specified in the block header. On return the highest RAM address loaded is contained in the index registers (X, Y).

$FFD8 — Save RAM to Device.

Communicating registers: X and Y index registers.
Error Returns: 5, 8 and 9
Stack Requirements:
Preparatory routines: $FFBA, $FFBD and FF9C.

Description: Saves memory from the bottom of memory (set by routine $FF9C) to the memory address (X, Y) to a logical device. A file name is not required for device 1 (the cassette deck) but an error condition exists for any other device saved without a file name. Device 0 (keyboard), and device 3 (screen) are not defined for this routine.

$FFDB — Set Real Time Clock.

Communicating registers: processor accumulator, X and Y index registers.
Error Returns: none.
Stack Requirements:
Preparatory routines: none.

Description: A system clock is maintained on a 1/60 second interrupt basis. Three bytes are provided to count jiffies up to 5,184,000 or 24 hours, at which point the clock rolls over to zero. To set the clock load the accumulator with the most significant, X index with the next most significant and Y index with least significant byte of time in jiffies.

$FFDE — Read Real Time Clock.

Communicating registers: processor accumulator, X and Y index registers.
Error Returns: none.
Stack Requirements: 2.
Preparatory routines: none.

Description: The system clock can be read at any time. Three bytes are returned containing a binary value corresponding to the time in 1/60 of a second. The accumulator contains the most significant, X index next most significant, and Y index the least significant byte.

$FFE1 — Check Stop Key.

Communicating registers: processor accumulator.
Error returns: none.
Stack Requirements.
Preparatory routines: none.

Description: This routine sets the Z' flag if the STOP key on the keyboard is pressed while the routine is called. All other flags are maintained. If the stop key is not pressed then the accumulator contains a byte corresponding to the last row of the keyboard scan. The user can check for other key closures in this manner.

$FFE4 — Get Character from Keyboard Queue.

Communicating registers: processor accumulator.
Error Returns: none.
Stack Requirements:
Preparatory routines: none.

Description: Get buffered character from keyboard queue. This subroutine removes one character from the keyboard queue and returns an ASCII value in the accumulator. If the queue is empty, the value returned will be zero. Characters are put into the queue by an interrupt driver scan which calls the routine $FF9F. Obviously these routines will not work if the interrupt is disabled in any way.

$FFE7 — Close All Files.

Communicating registers: none.
Error returns: none.
Stack Requirements: 11.
Preparatory routines: none.

Description: With this subroutine the pointers into the open file table are reset. Additionally, the routine $FFCC (close input and output channel) is called to reset the I/0 channels.

$FFEA — Increment Real Time Clock.

Communicating registers: none.
Error Returns: none.
Stack Requirements: 2.
Preparatory routines: none.

Description: Normally this routine is called every 1/60th second to

keep the system clock register updated. If the user processes own interrupts then this subroutine must be regularly called to update time and keep the STOP key routine functional.

$FFED — Return X, Y Organisation of Screen.

Communicating registers: X and Y index registers.
Error Returns:
Stack Requirements: 2
Preparatory routines: none.

Description: Returns the constant organisation of the screen e.g. 22 columns in .X and 23 lines in .Y. This routine has two main uses, it allows software to be written for the VIC 20 to be run on a future VIC 40 without any change in screen handling routines, the program will recognise which machine it is being run on. Secondly the 6561 allows the user to change the screen organisation, within certain limits, this routine can be used to check current organisation.

$FFF0 — Read/Set X, Y Cursor Position.

Communicating registers: X and Y index registers.
Error Returns: none.
Stack Requirements: 2.
Preparatory routines: none.

Description: A call with carry set reads the current X, Y position of the cursor on the screen into .X and .Y. A call with carry clear moves the cursor to location X, Y on the screen as determined by the contents of .X and .Y.

$FFF3 — Return Base Address of I/O.

Communicating registers: X and Y index registers.
Error Returns: none.
Stack Requirements: 2.
Preparatory routines: none.

Description: Returns the address of the page containing i/O in X, Y. This routine can be used with an offset to access memory mapped I/O devices in the VIC. This function and subsequent register accesses are machine dependent.

101

SYSTEM INITIALISATION AND AUTO POWER UP

When the VIC is switched on, a pre-defined initialisation sequence is executed. This initialisation sets the system up so that all RAM variables and vectors are correctly set; screen I/O and keyboard correctly defined; memory checked and the Basic interpreter set in the direct input mode. This initialisation sequence is triggered by the power on reset circuit. The reset circuitry on the VIC consists of a 555 timer 10 wired in such a way that when power is first switched on the reset line is held low for a short period. When the processor reset line is pulled low momentarily (minimum six clock cycles) it causes the processor to start execution of a program whose starting address is stored in locations $FFFC and $FFFD. The start routine whose address is contained in the reset sector is located at $FD22.

The start routine is typical of the great flexability inherant in the design of the VIC. It allows two options, go-to the initialisation routine contained in ROM expansion memory. The normal initialisation routine (located at $FD2F) is used whenever the VIC is to run programs in Basic or Basic programs with machine code subroutines. The initialisation code sets up the OS RAM vectors, the I/O devices, initialises the 6561 and then jumps to the start of Basic at location $C000.

The area of memory allocated for memory expansion on the VIC can be divided into three sections. Memory space reserved exclusively for ROM memory, space reserved for either ROM or RAM memory, and that reserved exclusively for RAM memory expansion. The section of expansion memory that is of interest in its connection with system initialisation is that reserved exclusively for ROM memory, locations $A000 to $BFFF. The first function of the start routine at $FD22 is to check if there is a ROM inserted in address space $A000. It does this by testing for a string of 5 characters starting at a specific location on the ROM. The sequence of five bytes searched for is:

Address — $A004	contents — $41	ASCII character — 'A'
$A005	$30	'O'
$A006	$C3	rvs 'C'
$A007	$C2	rvs 'B'
$A008	$CD	rvs 'M'

If the start routine finds this character string then program control jumps to an address stored in the first two bytes of the ROM, $A000 and $A001, the user written 'hard start' initialisation routine. A second jump address is stored in locations $A002 and $A003, this is

the 'warm start' routine which returns program control to Basic, it is called when the Restore key is pressed. If the 'AOCBM' character string is not found then the zero flag is set and the initialisation routine at $FD2F called.

This feature of the VIC will allow the machine to be used in a wide range of special applications where the programmer requires the machine to automatically power up into his program. All the commercial ROM based games packs use this method. The most interesting application is the enhancement or alteration of Basic by adding extra commands or changing the operation of existing commands. The example shown in Appendix 2 demonstrates how extra commands can be added, the example adds a range of graphic commands to Basic. This program can easily be modified to run any commands required by the user, simply by adding the command name and the start address of its associated subroutine into the command tables starting at $A056. Existing commands involving system I/O can be modified if those commands use one or more of the RAM vector addresses. This is done by simply changing the RAM vector address so that it points to a routine in the $A000 ROM, this routine then performs the new version of that routine. An example would be if the programmer required the VIC to communicate with devices using a different communications system, e.g. Centronics, to that provided on the VIC (serial IEEE, or RS-232). In this case all the I/O routines in the kernal would have to be changed so that data was input and output in the right format. The new versions of the I/O routines are put in the $A000 ROM and the initialisation routine simply sets up the correct new RAM vector addresses before returning to the main initialisation routine and Basic.

SYSTEM WEDGES

A system wedge is a machine code routine inserted into the normal system software, allowing the user to either modify. the system operation or monitor system functioning. There are two main positions where code can be wedged into the VIC system software, they are:

Interrupt wedge — code inserted into one or other of the two interrupt handling routines (NMI or IRQ) using the interrupt RAM vector jump addresses. A wedge routine inserted into the regular 60Hz IRQ interrupt can be used to scan for I/O input or perform background processing.

RAM vector wedge — a wedge routine inserted into one or more of the RAM vectors could be used to change the I/O configuration or file handling capabilities of the system.

CHARGOT wedge — this is a wedge inserted into the CHARGOT subroutine in page zero memory. Such a wedge can be used to intercept each Basic command in a program as it is executed. The principle use for such a wedge is to add extra commands to the Basic interpreter.

Three routines are required to implement a system wedge;

1 — the wedge must be initialised, this is done by a routine inserting a jump address to the wedge code into the CHARGOT routine or replacing an existing RAM vector address.

2 — the wedge code, this code performs the required function which amends or replaces the system function into which it is inserted. If an interrupt wedge is used, the wedge code should be terminated with a jump to the address originals contained in the interrupt vector. With a RAM vector wedge the wedge code can be terminated with either a jump to the normal function subroutine, or simply terminated with an RTS instruction, depending on the programmers requirements. A CHARGOT wedge is terminated with the section of the CHARGOT code which is replaced by the wedge jump address followed by an RTS instruction.

3 — when the wedge code is finished with it must be disabled by returning the vector addresses to normal or restoring the normal CHARGOT routine. The following system subroutines can be utilised to do this operation:

$E45B — restore vectors
$E3A4 — restore CHARGOT and zero page
Examples of these routines are shown in the following programs.

```
0073        ;Normal CHARGOT code
0073
0073    E6 7A        CHARGOT     INC #$007A
0075    D0 02                    BNE CHARGOT1
0077    E6 7B                    INC #$007B
0079    AD ** **     CHARGOT1    LDA POINTER
007C    C9 3A                    CMP #$3A
007E    B0 0A                    BCS CHARGOT2
0080    C9 20                    CMP #$20
0082    F0 EF                    BEQ CHARGOT
0084    38                       SEC
0085    E9 30                    SBC #$30
0087    38                       SEC
0088    E9 D0                    SBC #$D0
008A    60           CHARGOT2    RTS


0073        ;User wedge into CHARGOT code
0073
0073    E6 7A        WEDGE       INC #$007A
0075    D0 02                    BNE WEDGE1
0077    E6 7B                    INC #$007B
0079    AD ** **     WEDGE1      LDA POINTER
007C    C9 3A                    CMP #$3A
007E    F0 0A                    BEQ WEDGE2
0080    C9 20                    CMP #$20
0082    F0 EF                    BEQ WEDGE
0084    20 00 10                 JSR CODE
0087    20 00 18                 JSR REPLACE
008A    60           WEDGE2      RTS

1000        ;Start of user wedge program

1800        ;Replacement for CHARGOT code $84-$89
```

```
LOC    CODE          LINE


E378   20 5B E4      INIT    JSR INITV      ; GO INIT VECTORS
E37B   20 A4 E3              JSR INITCZ     ; GO INIT CHARGET & Z-PAGE
E37E   20 04 E4              JSR INITMS     ; GO PRINT INIT MESSAGE
E381   A2 FB                 LDX #STKEND-256 ; SET UP END OF STACK
E383   9A                    TXS
E384   4C 74 C4              JMP READY      ; GO TO READY




E387   E6 7A         INITAT INC CHRGET+7
E389   DO 02                 BNE CHDGOT
E38B   E6 7B                 INC CHRGET+8
E38D   AD 60 EA      CHDGOT LDA 60000
E390   C9 3A                 CMP #':
E392   BO OA                 BCS CHDRTS
E394   C9 20                 CMP #'
E396   FO EF                 BEQ INITAT
E398   38                    SEC
E399   E9 30                 SBC #'0
E39B   38                    SEC
E39C   E9 DO                 SBC #$DO
E39E   60            CHDRTS RTS
E39F   80                    BYT 128,79,199,82,88
E3A0   4F
E3A1   C7
E3A2   52
E3A3   58




E3A4   A9 4C         INITCZ LDA #76
E3A6   85 54                 STA JMPER
E3A8   85 00                 STA USRPOK
E3AA   A9 48                 LDA #<FCERR
E3AC   AO D2                 LDY #>FCERR
E3AE   85 01                 STA USRPOK+1
E3B0   84 02                 STY USRPOK+2
E3B2   A9 91                 LDA #<GIVAYF
E3B4   AO D3                 LDY #>GIVAYF
E3B6   85 05                 STA ADRAY2
E3B8   84 06                 STY ADRAY2+1
E3BA   A9 AA                 LDA #<FLPINT
E3BC   AO D1                 LDY #>FLPINT
E3BE   85 03                 STA ADRAY1
E3C0   84 04                 STY ADRAY1+1
E3C2   A2 1C                 LDX #INITCZ-INITAT-1
E3C4   BD 87 E3      MOVCHG LDA INITAT,X
E3C7   95 73                 STA CHRGET,X
E3C9   CA                    DEX
```

```
LOC    CODE          LINE

E3CA   10 F8              BPL MOVCHG
E3CC   A9 03              LDA #STRSIZ
E3CE   85 53              STA FOUR6
E3D0   A9 00              LDA #0
E3D2   85 68              STA BITS
E3D4   85 13              STA CHANNL
E3D6   85 18              STA LASTPT+1
E3D8   A2 01              LDX #1
E3DA   8E FD 01           STX BUF-3
E3DD   8E FC 01           STX BUF-4
E3E0   A2 19              LDX #TEMPST
E3E2   86 16              STX TEMPPT
E3E4   38                 SEC                ;READ BOTTOM OF MEMORY
E3E5   20 9C FF           JSR $FF9C
E3E8   86 2B              STX TXTTAB         ;NOW TXTAB HAS IT
E3EA   84 2C              STY TXTTAB+1
E3EC   38                 SEC
E3ED   20 99 FF           JSR $FF99          ;READ TOP OF MEMORY
E3F0   86 37       USEDEF STX MEMSIZ
E3F2   84 38              STY MEMSIZ+1
E3F4   86 33              STX FRETOP
E3F6   84 34              STY FRETOP+1
E3F8   A0 00              LDY #0
E3FA   98                 TYA
E3FB   91 2B              STA (TXTTAB)Y
E3FD   E6 2B              INC TXTTAB
E3FF   D0 02              BNE INIT20
E401   E6 2C              INC TXTTAB+1
E403   60          INIT20 RTS


E404   A5 2B       INITMS LDA TXTTAB
E406   A4 2C              LDY TXTTAB+1
E408   20 08 C4           JSR REASON
E40B   A9 36              LDA #<FREMES
E40D   A0 E4              LDY #>FREMES
E40F   20 1E CB           JSR STROUT
E412   A5 37              LDA MEMSIZ
E414   38                 SEC
E415   E5 2B              SBC TXTTAB
E417   AA                 TAX
E418   A5 38              LDA MEMSIZ+1
E41A   E5 2C              SBC TXTTAB+1
E41C   20 CD DD           JSR LINPRT
E41F   A9 29              LDA #<WORDS
E421   A0 E4              LDY #>WORDS
E423   20 1E CB           JSR STROUT
E426   4C 44 C6           JMP SCRTCH
```

107

```
LOC     CODE          LINE

E429    20 42         WORDS   .BYT ' BYTES FREE',13,0
E434    0D
E435    00
E436    93            FREMES .BYT 147,'**** CBM BASIC V2 ****'
E437    2A 2A
E44D    0D                    .BYT 13,0
E44E    00


E44F    3A C4         BVTRS   .WOR NERROR,NMAIN,NCRNCH,NQPLOP,
E451    83 C4                                       NGONE,NEVAL
E453    7C C5
E455    1A C7
E457    E4 C7
E459    86 CE
E45B                  ;
E45B    A2 0B         INITV   LDX #INITV-BVTRS-1 ; INIT VECTORS
E45D    BD 4F E4      INITV1 LDA BVTRS,X
E460    9D 00 03             STA IERROR,X
E463    CA                   DEX
E464    10 F7                BPL INITV1
E466    60                   RTS
```

3   THE 6561 VIDEO INTERFACE CHIP

Fig. 13 — Block diagram of the 6561 Video Interface chip.

111

OVERVIEW OF THE 6561 VIDEO INTERFACE CHIP

Many of the VIC's outstanding features are attributable to a single integrated circuit, the 6561 video interface chip. This single device provides all the circuitry necessary to generate colour programmable character graphics, with high screen resolution. The 6561 also incorporates sound effects generation, analogue to digital conversion for joysticks, and light pen capability. All these functions are under direct programmer control via the 16 addressable control registers of the 6561, to use these functions an understanding of the 6561's operation is essential. The video interface chip has three seperate functions, these are:

1 — Control and generation of the CRT display.
2 — Generation of the master oscillator clock.
3 — Specialised I/O for use in the video-games environment.

Only functions 1 and 3 are of interest to the programmer, the generation of the master oscillator clock is purely a hardware feature which ensures that all the system timing is synchronised with that of the 6561.

The control and generation of a colour display on a TV or monitor is the primary function of the 6561. To do this it must access four seperate areas of the VIC's memory space, the location and size of two of these areas is under programmer control. The four memory areas each have their own function in display generation, they are:

1 — Video RAM character pointer, each location corresponds to a position on the screen, location 0 in this section of RAM contains the ASCII code of the character displayed in the top left corner of the screen, location 1 has the next character to the right and so on for all the character positions on the screen. On the standard VIC with no memory expansion, this section of memory is 506 bytes long starting at location $1E00, if there is more than the bottom 3K memory expansion then the starting location is $1000.

2 — Colour pointer, this section of RAM is identical in size to the video character pointer and contains data on the foreground and background colour of each character. Location 0 in this section contains the foreground and background colour of the character in location 0 of the character pointer (i.e. top left character on the screen), and so on for all character positions on the screen. On the standard VIC with no memory expansion this section of memory is 506 bytes long starting at location $9600, if there is more than 3K of expansion memory then the starting address is $9800.

3 — Character generator, this section of memory contains the pattern of dots used to display each of the 255 different characters. the dot pattern for each character is contained in eight consecutive memory locations (this can be set to 16 consecutive locations if required), each bit of each byte corresponding to a dot position in that character, 1 = a dot and 0 = a space. The character generator in normal operation is stored in a 4K ROM, starting address $8000. By re-defining the character generator start address to point to 4K block of RAM a user definable character generator can be created.

4 — 6561 control registers, these registers control the way in which the 6561 operates and are located in 16 consecutive memory locations (their address is defined by hardware). The addresses used lie between $9000 and $900F.

In normal operation the kernal initialisation routines set up the registers of the 6561 to give the standard VIC display format, 23 lines by 22 columns, using the character generator at location $8000. The routines then put space characters into all locations in the character pointer RAM and set all locations in the colour pointer to give blue on white characters, control register 16 is set to give a white background and a cyan border.

To understand the operation of the VIC more completely, consider the diagram in Figure 8. This shows the three areas of memory used, video RAM, colour RAM, and character generator for a standard 22 column, 23 line display. Each of the 506 locations in the video RAM contains a code value or pointer into the character generator, in the diagram the location corresponding to column 22, line 10 contains the value 45. This means that character number 45 is displayed in that character space, the same location in the colour RAM contains the value 2, this makes the character red. Teh character number is used as an index into the character generator. The VIC fetches each of the 506 video RAM location values and performs an address computation on each of them to locate the desired value of the address of the eight bytes used to store each character in the character generator. The address computation is quite simple, if 8x8 characters are being used then the character code value (45 in the example) is multiplied by eight and the result added to the start address of the character generator (this base address is contained in the 6561 control register No.5.). The eight bytes of the character generator pointed to by this address, are transfered (one byte per scan line) via an internal shift

113

register on the 6561 to the video display as a serial bit stream. This bit stream combined with control pulses from the 6561 comprises the composite video output signal of the VIC. This signal is fed via a modulator to the TV set which then generates the required display.

Besides controlling the video output of the VIC the 6561 implements a series of interactive I/O features which are designed principaly for games applications. There are three of these features, they are:

1 — Sound generation system consisting of: three independently programmable tone generators, a white noise generator, and an amplitude modulator. The sound generation system can be used to create special sound effects and can even be used to play music of acceptable quality.

2 — Two anaologue to digital converters, these are intended for use with a potentiometer or joystick input, ideal for moving the cursor or games character about the screen.

3 — Light pen input, a photocell connected to this input and pointed to a part of the screen will return the screen co-ordinates of that point in two of the 6561 internal registers, ideal for interactive non keyboard input.

THE INTERNAL REGISTERS OF THE 6561

The sixteen eight bit control registers within the 6561 enable the microprocessor to control all the operating modes of the VIC. These control registers comprise sixteen successive memory locations starting at location 36864, and are accessible from either Basic (using PEEK and POKE) or machine code programs. The sixteen control registers of the 6561 are as follows:

Control register No. 1.
Location — Hex $9000   Decimal 35864
Contents in normal VIC operating mode — Decimal 12

Bits 0 to 6 of this register determine how far from the left side of the TV screen the first column of characters will appear. It is used to horizontally centre various sizes of video matrices on screen. Bit 7 when set to 1 enables the interlaced scan mode. Interlacing can be used with the appropriate hardware to display the VIC screen over a normal TV picture, this could be used for video titleing.

To demonstrate the horizontal movement of the screen by changing the contents of this register enter and run the following program:

```
10 FOR Q=0TO40
20 POKE 36864,Q
30 FOR X=0TO1000:NEXT X        :delay
40 NEXT Q
50 POKE 36864,12               : restore to normal
```

To demonstrate the effect of an interlaced display enter this command:

```
POKE 36864,140
```

Control register No. 2.
Location — Hex $9001   Decimal 36865
Contents in normal VIC operating mode — Decimal 38

Determines how far from the top of the TV screen the first row of characters will appear. It is used to vertically centre various sizes of video matrix on the screen.

To demonstrate the vertical movement of the screen by changing the contents of this register enter and run the following program:

```
10 FOR Q=0TO150
20 POKE 36865,Q
30 FOR X=0TO100:NEXT X        :delay
```

```
40 NEXT Q
50 POKE 36865,38                    :restore to normal
```
Control register No. 3.
Location — Hex $9002   Decimal 36866
Contents in normal VIC operating mode — Decimal 150

Bits 0-6 determine the number of columns in the video matrix, thus for a 22 column screen bits 0-6 will contain the value 22. Bit 7 is part of the video matrix address stored in control register No. 6, this bit is normally set to logical '1' (i.e. add decimal 128 to value in bits 0-6).

To demonstrate the use of this register to change the number of columns displayed on the screen enter and run the following program:
```
10 FOR Q=128 TO 155
20 POKE 36866,Q
30 FOR X=0TO1000:NEXT X          :delay
40 NEXT Q
50 POKE 36866,150               :restore to normal
```

Control register No. 4.
Location — Hex $9003   Decimal 36867
Contents in normal VIC operating mode — Decimal 174

Bits 1 to 6 set the number of rows in the video matrix. Bit 0 is used to select either 8x8 characters (bit 0 = 0) or 16x8 character matrices (bit 0 = 1). Bit 7 is the least significant bit of the raster line number found in control register No. 5.

To demonstrate the use of this register to change the number of lines displayed on the screen enter and run the following program:
```
10 FOR Q=128 TO 180 STEP 2
20 POKE 36867,Q
30 FOR X=0TO1000:NEXT X          :delay
40 NEXT Q
50 POKE 36867,174               :restore to normal
```

Control register No. 5.
Location — Hex $9004   Decimal 36868
Contents in normal VIC operating mode — Variable

This register contains the number of the line currently being scanned by the TV raster beam.

Control register No. 6.
Location — Hex $9005   Decimal 36869
Contents in normal VIC operating mode — Decimal 240

Bits 0 to 3 determine the starting address of the character cell space
(note that these bits form lines A13 to A10 of the actual address). Bits
4 to 7 together with bit 7 of control register No. 3., determine the
starting address of the video matrix (these bits form address lines A13
to A9 of the actual address).

Control register No. 7.
Location — Hex $9006   Decimal 36870
Contents in normal VIC operating mode — Decimal 0

Contains the latched horizontal position of the light pen

Control register No. 8.
Location — Hex $9007   Decimal 36871

Contents in normal VIC operating mode — Decimal 0

Contains the latched vertical position of the light pen

Control register No. 9.
Location — Hex $9008   Decimal 36872
Contents in normal VIC operating mode - Decimal 255

Contains the digitised value of input on potentiometer No. 1., (see
section on joysticks for details on operation and use).

Control register No. 10.
Location — Hex $9009   Decimal 36873
Contents in normal VIC operating mode — Decimal 255

Contains the digitised value of input on potentiometer No. 2., (see
section on joysticks for details on operation and use).

Control register No. 11.
Location — Hex $900A   Decimal 36874
Contents in normal VIC operating mode — Decimal 0

Bits 0 to 6 set the frequency of the first audio oscillator.
Bit 7 turns the oscillator on (=1) or off (=0).

Control register No. 12.
Location — Hex $900B   Decimal 36875
Contents in normal VIC operating mode — Decimal 0

Bits 0 to 6 set the frequency of the second audio oscillator. Bit 7 turns
the oscillator on (=1) or off (= 0).

Control register No. 13.
Location — Hex $900C   Decimal 36876
Contents in normal VIC operating mode — Decimal 0

Bits 0 to 6 set the frequency of the third audio oscillator. Bit 7 turns the
oscillator on (=1) or off (=0).

Control register No. 14.
Location — Hex $900D   Decimal 36877
Contents in normal VIC operating mode — Decimal 0

Bits 0 to 6 set the base frequency for the pseudo white noise
generator. Bit 7 turns the noise generator on (=1) or off (=0).

Control register No. 15.
Location — Hex $900E   Decimal 36878
Contents in normal VIC operating mode — Decimal 0

Bits 0 to 3 set the volume of the composite audio signal (note that at
least one sound generator must be turned on for any sound to be
produced). Bits 4 to 7 contain the auxiliary colour code used in
conjunction with the 'Multicolour mode' of operation.

Control register No. 16.
Location — Hex $900F   Decimal 36879
Contents in normal VIC operating mode — Decimal 27

Bits 4 to 7 select one of sixteen colours for the background common to
all characters on the screen (essentially they set the colour of the
background area within the video matrix). Bits 0 to 2 select one of
eight colours for the exterior border area of the screen, this is the area
outside the video matrix. Bit 3 determines whether the video matrix is
to be displayed as different coloured characters on a common
background colour (bit 3=1), or inverted (bit 3=0) where all

characters have the same colour, but different background colours determined by the code in the colour RAM. Bit 3 has no effect when the 'Multicolour' mode is selected on the 6561, the other functions of control register No. 16., also vary in this mode.

To demonstrate the changing of the border colour (there are eight different colours) by bits 0-2, run the following program. Note that the screen colours are retained in their normal mode of blue characters on a white background.

```
10 FORQ=0TO7
20 PIKE36879,Q+24              :change border colour
30 FORX=1TO1000:NEXTX          :delay
40 NEXTQ
50 GOTO10
```

The sixteen different background colours are selected by bits 4-7 and the following program demonstrates the changing of the background colour, note the cyan border colour and the blue character colour remain unchanged.

```
10FORQ=0TO255STEP16
20 POKE36879,Q+11              :new background colour

30 FORX=1TO1000:NEXTX
40 NEXTQ
50 GOT10
```

Bit 3 controls whether characters are displayed on a common background colour, or inverted so that all characters are the same colour but the background is a different colour. The following short program demonstrates this.

```
10 POKE36879,PEEK(36879)AND247    :invert background
20 FORQ=-TO1000:NEXTQ             :delay
30 POKE36879,PEEK(36879)OR8       :restore to normal
40 FORQ=0TO1000:NEXTQ
50 GOTO10
```

119

6561 DISPLAY MODES

The VIC has two display modes, normal text mode and user definable character mode. The modes are determined by the position in memory of the character generator. There are also two modes of colour operation, high resolution, and multicolour. The VIC is thus capable of several permutations of colour and display mode.

The two display modes depend on whether the normal internal ROM based character generator is used or a user definable RAM character generator. The position of the character generator within processor memory space is determined by the contents of bits 0-3 of control register No.5. These four bits form bits A10 to A13 of the actual character generator address as follows:

```
┌─┬─┐   ┌─┬─┐
│3│2│   │1│0│
└─┴─┘   └─┴─┘
 1  1    0  1  x x    x x x x    x x x x
 $  3       4            0          0
```
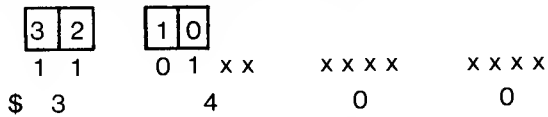
The normal contents of bits 0-3 of control register No. 5., are zero, the way the VIC is configured, this gives a character generator address of Hex $8000 (decimal 32768). Starting at this location is a 4K ROM, the character generator. This contains the actual dot pattern for each of the 256 different characters which can be displayed. The 4K character generator ROM actualy contains two seperate character generators each occupying 2K of ROM. The first of these two character generators which starts at address Hex $8000 (decimal 327680 contains the dot pattern for the 128 normal upper case and graphics characters plus the 128 reverse field versions of the same characters. The second character generator starts at location $8800 (decimal 34816) and is identical to the first except that part of the graphics character set is replaced by lower case characters. When the second character set is enabled the VIC will normally display in lower case characters rather than the normal upper case, upper case will be displayed with the shift key depressed. The second character generator can be enabled normally, by pressing the shift key and the Commodore logo key simultaneously. Alternatively one can change the contents of control register No. 5., thus:

```
POKE 36869,242      :set lower case display mode
POKE 36869,240      :set upper case display mode
```

This simply shifts the starting address of the character generator up 2K in memory thereby accessing the second character generator.

The character generator starting address in control register No. 5. can be changed so that the character generator is located in RAM, thereby allowing user definable characters to be created. The starting address of the user definable RAM character generator on the VIC can be any 2K (4K if 8x16 characters are used) block or RAM, located between address Hex $1000 and $3000. It should be located at the highest possible address, and protected from being overwritten by Basic by lowering the top of memory pointers, to protect the RAM space used by the character generator. The setting up of control register No. 5., has the following rules:

1 — The starting address is always located at the beginning of a 1K block.

2 — If the contents of bits 2 and 3 are both zero then the starting address defaults to the ROM at $8000 plus the offset stored in bits 0 and 1, this offset is in increments of 1K.

3 — Bits 2 and 3 contain the starting address in increments of 4K.

Thus to put the user definable character generator to start at 11K up in memory, — Hex $2000 — 2x 4K block plus 3x 1K block — then bits 0 to 3 would be set up as follows:

Bits	3	2	1	0
Binary contents	1	0	1	1
representing	2x 4K blocks		3x 1K blocks	

The user definable character generator is very important, since it not only allows special graphics characters to be created, but it also allows high resolution point plotting on the VIC. This allows a graph or display to be created with a resolution of 176 points in the horizontal by 184 points vertically, sufficient to give a very good quality display. High resolution point plotting is achieved by programming techniques using the user definable character generator. The use of the RAM character generator must be understood before these techniques can be explained.

The first stage in creating a user definable character set, is to allocate a block of RAM memory for storage of the character generator. If characters on an 8x8 matrix are being displayed then 2048 memory locations are required, if an 8x16 matrix is to be used then 4096 locations are required. Since a standard VIC has only 3584
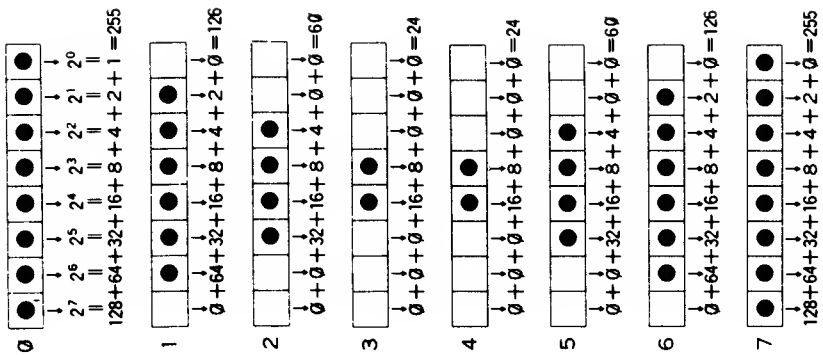
Fig. 14 — Conversion of a character into numerical values.
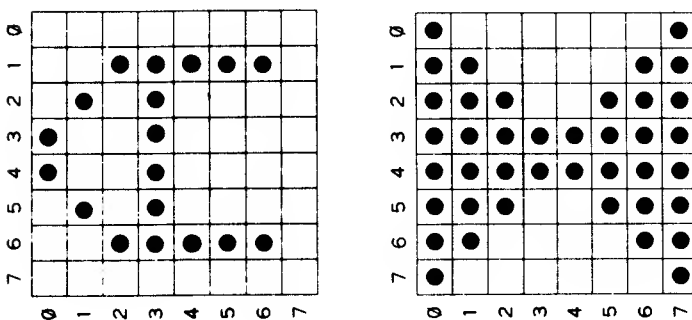
Fig. 15 — Examples of layout in design of characters.

122

RAM memory locations available to the user, an 8x8 matrix user definable character generator using 2048 of these locations is the only one feasible. The user RAM on a standard unexpanded VIC starts at memory address 4096 and goes on to address 7679. The character generator can be programmed to start at any of the following addresses within that range; 4096, 5120, 6144, or 7168. Since 2048 locations are required for the character generator the only possible starting location is 5120, this leaves 1024 bytes free for user programs (not much, purchase of the standard 3K RAM expansion module is strongly recommended, it's use will not change the start address recommended above). This area of RAM chosen for use by the character generator must be protected from being overwritten by a Basic program or data. If this happened the display would be destroyed. The user definable character generator can be protected from being overwritten by lowering the top of memory pointers, thus:

10 POKE 51,255 : POKE 52,19
11 POKE 55,255 : POKE 56,19
12 CLR

The next stage is to put the data on each character into the new character generator, by using POKE commands or machine code load statements to put information into the 2048 memory locations. Before this can be done each of the new characters must be designed, this entails drawing each character on an 8x8 grid, see Fig.15. Once the character has been designed it can be converted into the block of eight numerical values for storage in the character generator. Each line in the 8x8 grid corresponds to a byte of data, and each of the eight bits in that byte corresponds to a dot or column position on that line. Information is stored in memory in binary, thus by considering each bright dot to be a logical '1' and each space a logical '0', a line of dots in each character can be converted into a numerical value, the way this is done is shown in Fig.14. Some examples of character designs and their conversion to numerical values are shown in Fig.16. From these values a table can be created, one column having the character generator address, and the corresponding entry in the second column having the value to be put into that location. The table is divided into blocks of eight entries, each block containing the data for one character. Each of these blocks of eight entries is numbered starting at 0 and going up to 255. These numbers correspond to the ASCII or character code numbers stored in the video RAM when the characters are displayed. An example table using the character designs in Fig.15, is shown in Fig.

123

	7	6	5	4	3	2	1	0	
0									→0+0+0+0+0+0+0+0=0
1				●	●				→0+0+0+16+8+0+0+0=24
2			●	●	●	●			→0+0+32+16+8+4+0+0=60
3		●	●	●	●	●			→0+64+32+16+8+4+2+0=126
4	●	●	●	●	●	●	●	●	→128+64+32+16+8+4+2+1=255
5				●	●				→0+0+0+16+8+0+0+0=24
6			●			●			→0+0+32+0+0+4+0+0=36
7		●					●		→0+64+0+0+0+0+2+0=66

	7	6	5	4	3	2	1	0	
0				●	●				→0+0+0+16+8+0+0+0=24
1				●		●			→0+0+0+16+0+4+0+0=20
2				●		●			→0+0+0+16+0+4+0+0=20
3				●			●		→0+0+0+16+0+0+2+0=18
4			●	●					→0+0+32+16+0+0+0+0=48
5		●	●	●					→0+64+32+16+0+0+0+0=112
6		●	●						→0+64+32+0+0+0+0+0=96
7									→0+0+0+0+0+0+0+0=0

```
5120 —   0  ⎫
5121 —  24  ⎪
5122 —  60  ⎪
5123 — 126  ⎪
5124 — 255  ⎬  Character # 1
5125 —  24  ⎪
5126 —  36  ⎪
5127 —  66  ⎭

5128 —  24  ⎫
5129 —  20  ⎪
5130 —  20  ⎬  Character # 2
5131 —  18  ⎪
5132 —  48  ⎪
etc.        ⎭
```

Fig. 16 — Conversion of user characters into a character generator table.

16. The table need only contain the number of characters actually required, all 255 possible character blocks do not have to be filled in. It is advisable that the table starts at the first location in the character generator, any gaps left should be filled with zeros. If the character generator is being loaded from a Basic program then the values in the tables are best stored as DATA statements, these values are then entered into memory using POKE commands, thus:

```
20 FOR I = 0 TO 2048
21 READ A
22 IF A = "*" THEN 30
23 POKE 5120 + I,A
24 NEXT
30 END

100 DATA 24, 20, 20, 28, 48, 112, 96, 0
110 DATA 0, 24,60, 126, 255, 24, 36, 66
120 DATA 255, 126, 60, 24, 24, 60, 126, 255
130 DATA *
```

In the majority of applications alphanumeric characters are required in addition to user defined graphics characters, in such cases part of the data in the ROM based character generator must be transferred to the new RAM character generator. All the alphanumeric characters plus the VIC graphics characters (or lower case depending on which of the two character generators is accessed) are contained in the first 128 characters of the character generator, the remaining 128 characters are the reverse field versions of the first 128 characters. The first 128 characters of the ROM character generator are transfered to the new RAM character generator using a combination of PEEK and POKE commands thus:

```
20 FOR I=0 TO 1024
30 POKE 5120+I,PEEK(32768+I)
40 NEXTI
```

This leaves 128 possible user definable characters starting at address 6155, these characters can be filled as described above, and will have an ASCII code starting value of 128. An example of the routine to enter the character generator data will be as follows:

```
20 FOR I=0 TO 1024
21 POKE 5120=I,PEEK(32768+I)
22 NEXTI
```

```
30 FOR I = 0TO 1024
31 READ A
32 IF A="*" THEN 200
33POKE 6144=I,A
34 NEXT

 60 REM DATA FOR ASCII CODE CHARACTERS 128, 129, 130
100 DATA 24, 20, 20, 18, 48, 112, 96, 0
110 DATA 0, 24, 60, 126, 255, 24, 36, 66
120 DATA 255, 126, 60, 24, 24, 60, 126, 255
130 DATA *
```

Having loaded the user definable character generator it can be used, it will remain in the VIC until the machine is switched off and can thus be used by more then one program. To use the RAM character generator two of the 6561 registers must be changed, thus:

```
200 POKE 36869, 253
210 POKE 36866, PEEK(36866)OR128
```

Once the user definable RAM character generator has been set up and the 6561 registers changed to utilise the new character generator it can be used to generate special displays. If POKE commands are used to place the characters in the video RAM memory then the ASCII code value of the new characters is used. If the new characters are incorporated into strings then it is essential to know which character in the normal character set the new character replaces. This can be determined by using the table of VIC ASCII codes and looking for the character with the same code value as the new character. When the program is written the normal characters are inserted into the string, when the program is run they will be automatically replaced by the new characters. It is important to note that when using POKE commands, the colour RAM location corresponding to the location where the character is to be displayed must also be set to give the required colour, otherwise the display will be white on white and therefore invisible. To restore the normal function of the VIC ROM character generator use the following two lines:

```
500 POKE 36869,240
510 POKE 36866,150
```

High resolution point plotting uses exactly the same principles as the generation of user definable characters. It entails filling the video RAM with each of the 255 character codes (only half the screen can be used with 8x8 characters). The RAM character generator can then be

used as a high resolution memory mapped display. If all bytes in the RAM character generator are set to zero then the screen is blank. Set one bit in one of the characters and a single high resolution dot will appear on the screen. The relationship between a single dot on the screen, the locations in the RAM character generator, and the code value in each of the video memory locations is shown in Figure 17. Showing that the basis of high resolution plotting is simply filling the video RAM corresponding to the screen area of the high resolution display with successive and incremented code values. The rest is a matter of calculation to ensure that the correct bits are set in each of the eight bytes corresponding to each of the character codes used in the video RAM. A high resolution plotting program consists of two parts, the initialisation and the point plot subroutine. The initialisation sets up the registers of the 6561 for a user definable character generator, lowers the top of memory to protect that character generator, puts the correct data into the video and colour RAMs and clears the contents of the RAM character generator. The point plot subroutine is called whenever a point is to be plotted or erased, and consists of a routine which calculates from given X and Y co-ordinates which bit in which byte of the RAM character generator is to be set or erased. It should be noted that the area of the screen devoted to high resolution plotting can vary from just a few adjacent character spaces to the whole screen (to do this the 6561 is initialised to display 8x16 characters rather than the normal 8x8, this requires the RAM character generator to be enlarged to 4K. An example of a set of Basic routines to plot points in high resolution, plus lines and circles, is contained in the following program (these routines use a 2K character generator and 8x8 characters so the display occupies only half the screen, the 6561 registers have been used to centre the display).

127
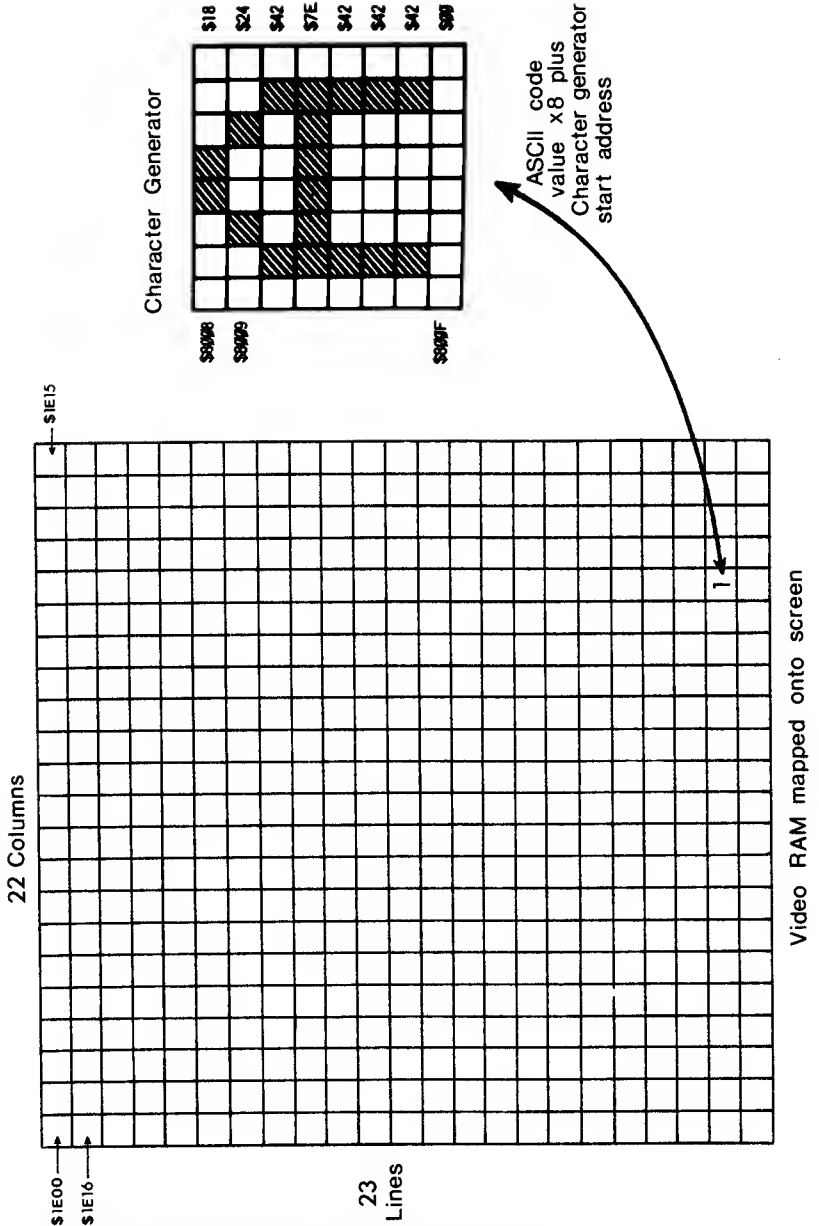
Fig. 17 — Relationship between the character generator, video matrix, and the displayed character.

128

```
1 REM ********************************************
2 REM *PROGRAM TO PLOT THE GRAPH OF A FUNCTION
3 REM *IN HIGH RESOLUTION ON THE VIC
4 REM ********************************************
5 REM
6 REM * INITIALISE 6561 REGISTERS
7 PRINT"◤◢"
8 POKE36867,128
9 POKE36865,60
10 F(8)=0:F(0)=128:F(1)=64:F(2)=32:F(3)=16
20 F(4)=8:F(5)=4:F(6)=2:F(7)=1
30 FORQ=0TO255
32 POKE7680+Q,0
34 POKE38400+Q,2
36 NEXTQ
40 FORQ=5120TO5120+255*8
42 POKEQ,0
44 NEXTQ
45 POKE36869,253
46 POKE36866,PEEK(36866)OR128
47 POKE36867,150
60 REM
61 REM *PLOT GRAPH OF FUNCTION IN LINE 90
62 REM
80 FORC=0TO175
90 L=45+40*SIN(C/10)
91 REM
92 REM *HIGH RESOLUTION POINT PLOT ROUTINE
93 REM
100 A=5120
110 LR=L/8
120 LA=INT(LR)
130 A=A+(LA*176)
140 LR=(LR-LA)*8
300 CR=C/8
310 CA=INT(CR)
320 A=A+(CA*8)
325 A=A+LR
330 CR=INT((CR-CA)*8)
400 POKEA,PEEK(A)ORF(CR)
```

```
500 NEXTC
550 REM
551 REM *WAIT FOR KEY PRESS THEN RETURN
552 REM *SCREEN TO NORMAL.
553 REM
600 GETA$:IFA$=""THEN600
1000 POKE36869,240
1010 POKE36866,150
1020 POKE36867,174
1030 POKE36865,38


1 REM *********************************
2 REM *PROGRAM TO PLOT HIGH RESOLUTION
3 REM *POINTS, LINES AND CIRCLES ON THE VIC.
4 REM *********************************
5 REM
6 REM *INITIALISE 6561 AND CHAR GEN
7 REM
8 POKE36867,128
9 POKE36865,60
10 F(8)=0:F(0)=128:F(1)=64:F(2)=32
20 F(3)=16:F(4)=8:F(5)=4:F(6)=2:F(7)=1
35 FORQ=0TO255
37 POKE7680+Q,Q
38 POKE38400+Q,2
39 NEXTQ
40 FORQ=5120TO5120+255*8
41 POKEQ,0
42 NEXTQ
45 POKE36869,253
46 POKE36866,PEEK(36866)OR128
47 POKE36867,150
90 REM
91 REM *DATA FOR LINE DRAWING
92 REM *START AT COORDINATES X1,Y1
93 REM *END AT COORDINATES X2,Y2
94 REM
100 READX1,Y1,X2,Y2
```

```
105 IFX1=255THEN200
110 GOSUB1000
120 GOTO100
150 DATA 80,10,100,40
151 DATA 80,10,60,40
152 DATA 95,38,95,80
153 DATA 65,38,65,80
154 DATA 65,80,95,80
155 DATA 85,80,85,60
156 DATA 90,80,90,60
157 DATA 85,60,90,60
158 DATA 70,75,70,60
159 DATA 75,75,75,60
160 DATA 70,75,75,75
161 DATA 70,60,75,60
162 DATA 70,50,70,35
163 DATA 75,50,75,35
164 DATA 70,50,75,50
165 DATA 70,35,75,35
166 DATA 85,50,85,35
167 DATA 90,50,90,35
168 DATA 85,50,90,50
169 DATA 85,35,90,35
170 DATA 20,80,20,50
171 DATA 22,80,22,50
172 DATA 120,80,120,50
173 DATA 122,80,122,50
188 REM *END OF LINE DATA
189 DATA 255,255,255,255
190 REM
191 REM *DATA FOR DRAWING CIRCLES
192 REM *CENTRE AT COORDINATES CX,CY
193 REM *RADIUS R
194 REM
199 DATA 255,255,255,255
200 CX=21:CY=40:R=10
210 GOSUB3000
220 CX=121:CY=35:R=15
230 GOSUB3000
240 GETA$:IFA$=""THEN240
```

```
1000 REM
1010 REM *LINE DRAWING ROUTINE
1020 REM *USES DATA FROM LINE DATA TABLE
1030 REM
1200 XD=X2-X1
1210 YD=Y2-Y1
1230 A0=1:A1=1
1240 IFYD<0THENA0=-1
1250 IFXD<0THENA1=-1
1270 XE=ABS(XD):YE=ABS(YD):D1=XE-YE
1280 IFD1>=0THEN1320
1290 S0=-1:S1=0:LG=YE:SH=XE
1300 IFYD>=0THENS0=1
1310 GOTO1340
1320 S0=0:S1=-1:LG=XE:SH=YE
1330 IFXD>=0THENS1=1
1340 REM
1350 TT=LG:TS=SH:UD=LG-SH:CR=LG-SH/2
1355 D=0
1360 REM
1370 C=X1:L=Y1:GOSUB2100
1380 IFCT>=0THEN1420
1390 CT=CT+TS:X1=X1+S1:Y1=Y1+S0
1410 GOTO1460
1420 CT=CT-UD:X1=X1+A1:Y1=Y1+A0
1460 TT=TT-1
1470 IFTT<0THENRETURN
1480 GOTO1370
2000 REM
2010 REM *POINT PLOT ROUTINE
2020 REM *USED BY LINE AND CIRCLE DRAW
2030 REM *ROUTINES
2040 REM *C=X COORDINATE
2050 REM *L=Y COORDINATE
2060 REM
2100 A=5120
2110 LR=L/8
2120 LA=INT(LR)
2130 A=A+(LA*176)
2140 LR=(LR-LA)*8
```

```
2300 CR=C/8
2310 CA=INT(CR)
2320 A=A+(CA*8)
2325 A=A+LR
2330 CR=INT((CR-CA)*8)
2400 POKEA,PEEK(A)ORF(CR)
2500 RETURN
2600 GETA$:IFA$=""THEN2600
3000 REM
3001 REM *CIRCLE DRAWING ROUTINE
3002 REM *OX AND OY ARE OFFSET VARIABLES
3003 REM *WHICH DETERMINE WHETHER A CIRCLE
3004 REM *OR ELIPSE IS DRAWN
3005 REM
3010 OX=1:OY=1.2
3020 A=2*π
3030 N=100
3040 INC=(A-0)/N
3050 FORI=0TOASTEPINC
3060 X=R*SIN(I):X=INT(X*OX+CX+.499)
3070 Y=R*COS(I):Y=INT(Y*OY+CY+.499)
3080 L=Y:C=X:GOSUB2100
3090 NEXTI
3100 RETURN
```

DISPLAY FORMAT CONTROL

The standard display format of the VIC is a video display with 22 characters horizontally and 23 lines vertically, with each character consisting of a matrix of 8x8 dots. All these values, together with the position of the text area in the screen, can be changed by altering the contents of specific registers of the 6561. There are five screen format variables which can be changed by the user, they are:

1 — Position of the first column of characters from the left hand side of the screen. This can be changed by altering the contents of control register No. 1., at location 36864, the normal value in this register is 12. By increasing the value in the register by two the position of the first column of characters is moved to the right by one character space. The minimum value in 36864 is 0, this puts column 5 of the display area on the left hand edge of the screen, the maximum value depends on the screen width and varies from 22 with a screen width of 22 character to 64 with a screen width of 1 column.

2 — Position of first row of characters from the top of the screen. This can be changed by altering the contents of control register No. 2., at location 36865, the normal value in this register is 38. By increasing the value in the register by four the display area is moved down from the top of the screen by one line. The minimum value in 36865 is 0 and the maximum value 255, a value of greater than 130 will cause the display to disappear off the bottom of the screen and can be used as a means of screen blanking.

3 — Determine the number of rows in the display. The number of rows displayed is determined by the value stored in bits 1 to 6 of control register No. 4., located at 36867, the normal value in this register is 174. The value stored in this register is obtained by multiplying the desired number of rows by two and adding the result to the value 128. The minimum number of rows is 1 (register value 130) and the maximum number of rows displayable is 32, — this is only achievable if the screen width is reduced so as not to exceed the 506 bytes in the video RAMs (register value for 32 rows is 192). Changing the number of rows in the display to other than 23 will disrupt the operation of the screen editor.

4 — Determine the number of columns in the display. The number of columns displayed is determined by the value stored in bits 1 to 6 of control register No. 3., located at 36866, the normal contents is 150.

The value stored in this register is obtained by adding the minimum number of columns desired to the value 128. The minimum number of columns is 1 (register value 129), and the maximum number is 27 (register value 155) this is only achievable if the number of rows is reduced so as not to exceed the 506 bytes in the video RAMs. Changing the display width to a value other than 22 characters will disrupt the operation of the screen editor.

5 — Determine the size of each individual character matrix. All characters are normally displayed as a matrix of 8x8 dots on the screen, but changing the value in bit 0 of register No. 4., location 36867, allows this to be changed to an 8x16 matrix. Add 1 to the current contents of this register and the character size will be doubled so that it is 16 dots high and 8 dots wide. Return to normal by subtracting one. The larger size character matrix is required in high resolution point plotting·in order to fill the whole screen with the 255 characters of the character generator memory.

VIDEO MEMORY ADDRESS CONTROL

Three areas of processor memory space are required by the 6561 in addition to the memory locations occupied by the 6561 control registers. These three memory areas are video RAM, colour RAM, and character generator. Of these three the location of the video RAM and the character generator are variable and under control of registers in the 6561. The starting locations of these two blocks of memory are stored in control register No. 6., location 36869, plus bit 7 of control register No. 3., location 36866. Both addresses are stored in register No. 6. as the most significant four bits of a 14 bit address, bit 7 of control register No. 3., is address line 9 of the video RAM address. Their use is best illustrated by the following sample:

Control register No. 3.   1 x x x x x x x

Control register No. 6.   0 0 0 0 1 1 0 1

14 bit video RAM address is:

	CR No. 6 bits		CR No. bit	
	7 6	5 4 7	x x x x	x x x x
binary —	0 0	0 0 1 x	x x x x	x x x x
hex —	0	2	0	0

14 bit character generator address is:

	CR No. 6 bits			
	3 2	1 0		
binary —	1 1	0 1 x x	x x x x	x x x x
hex —	3	4	0	0

In this example the video RAM is located at Hex $0200 and the character generator at $3400. The starting position is incremented in jumps of 1K for the character generator and 512 bytes for the video RAM. The addressing range of both the character generator and the video RAM are both limited since to access all the processor memory space requires a16 bit address. This limitation is partly overcome by using hardware addressing, thus the character generator start address defaults to $8000 when the contents of all four address bits in CR No. 6 which control this address are zero.

COLOUR CONTROL

The VIC has two modes of colour operation, 'High resolution' mode and 'Multicolour' mode. The operating mode employed plus the colours used are determined by the contents of control registers No. 15, and No. 16., of the colour video RAM. The colour video RAM is located in a 506 byte block or memory starting at location $9600 (decimal 38400), if there is more than 8K of user memory then the starting location of colour RAM moves down to $9400 (decimal 37888). The colour video RAM is only four bits wide, bits 0-2 are used to select the character colour and bit 3 is used to determine if that character is in 'high resolution' or 'multicolour' mode.

The 'High resolution' mode is selected by having bit 3 of the video colour RAM set to zero, this is the normal mode of operation. In this mode there is a one to one correspondence between character generator bits and the dots displayed on the screen. This means that all 'one' bits will be displayed as dots of one colour and all 'zero' bits as dots of another colour. Each character has two colours, a foreground (all the 'one' bits) and a background colour (all the 'zero' bits). One of these colours is determined by the first three bits of the video colour RAM and the other by bits 4-7 of control register No. 16. In normal operation the foreground colour is stored in the video colour RAM and the background colour which is common to all characters displayed on the screen is stored in register No. 16. This can be reversed so that all characters have the same foreground colour which is determined by register No. 16., and different background colours set by the contents of the colour video RAM. Whether a common foreground or a common background is selected depends on the contents of bit 3 of control register No. 16. If bit 3 is set to 1 then the display will have different coloured characters on a common background colour, if bit 3 = 0 then all characters will have the same colour against a different colour background. In addition to the foreground and background colours the 6561 allows the colour of the border around the display area to be changed, this is selected by bits 0-2 of control register No. 16.

The colours which can be displayed on the VIC are divided into two groups. The first group has eight colours, these colours can be used for the foreground or video colour RAM stored colour, and the border. The second group has sixteen colours which can be used for the background colour, (stored in control register No. 16.). and for the auxiliary colour (this is only used in the 'Multicolour' mode). The colours available in each of the the groups are as follows:

	Auxiliary/Background Colours	Border/Character Colours
0	Black	Black
1	White	White
2	Red	Red
3	Cyan	Cyan
4	Magenta	Magenta
5	Green	Green
6	Blue	Blue
7	Yellow	Yellow
8	Orange	
9	Light Orange	
10	Pink	
11	Light Cyan	
12	Light Magenta	
13	Light Green	
14	Light Blue	
15	Light Yellow	

In summary: in 'High resolution' mode the colours used for a particular character are:

1 — Set bit 3 of register No. 16., for common background or common foreground.
    common foreground — POKE 36879,PEEK(36879)AND247
    common background — POKE 36879,PEEK(36879)OR8

2 — Set the common background/foreground colour in bits 4-7 of control register No. 16. There are sixteen possible colours, it is the colour number as shown in the above table which is stored in the register, as in the following example where variable C is the colour and is set to a value between 0 and 15:
    POKE 36879,PEEK(36879)AND15
    POKE 36879,PEEK(36879)OR(C*16)

return to normal with — POKE 36879,27

3 — Set the border colour in bits 0-2 of control register No. 16. There are 8 possible border colours and it is the colour number shown in the above table which is stored in the register, as in the following example where variable C is the colour and is set to a value between 0 and 7:

POKE 36879,PEEK(36879)AND248
POKE 36879,PEEK(36879)OR O

4 — Put the colour code for each character to be displayed into the corresponding location in the colour video RAM. There are eight possible character colours, see above table, they are stored in bits 0-2 of the 506 locations in the colour video RAM. This is done automatically in a PRINT statement where the character colours can be embedded in the string as colour commands. If POKE commands are used to put characters into the video RAM then the colour code must also be POKEd into the corresponding location in the colour RAM. Given the column number — COL, and the line number – LIN, of the display plus the ASCII code of the character — A, and the colour code for that character — C, the following routine will put the character and its colour into the correct locations in the two video RAMs:

```
100 Q = LIN*22+COL
110 POKE 38400+Q,C
120 POKE 7680+Q,A
```

The 'Multicolour' mode is selected by having bit 3 of the video colour RAM set to one. In this mode there is a two to one correspondence between character generator bits and the dots displayed on the screen. This means that two bits of the character generator matrix for that character code correspond to one dot on the screen, and the colour of that dot is determined by the two bit code in the character generator. Unlike the 'High resolution' mode in which only two colours can be displayed for each character, 'Multicolour' mode allows four colours per character. However, since two bits of character generator data correspond to a single dot on the screen the horizontal resolution is half that of the 'high resolution' mode. That is each 8x8 character cell in memory maps onto an 8x4 character on screen (8 lines of 4 dots). Each character occupies the same space in either mode since both modes can be intermixed in a display, meaning that a single dot in 'Multicolour' mode occupies the same space as two horizontal dot positions in the 'High resolution' mode. The amount of memory required for storage of the 8x4 'multicolour' characters is the same as that required for the 8x8 characters, the data is simply mapped differently on screen.

The 'Multicolour' mode is not suitable for use with the ROM based character generators but can be very effective when used with a user

definable RAM character generator. This is because the ROM character generators are designed for 'High resolution' mode displays where each bit in the character matrix represents a dot position on the screen. In 'Multicolour' mode the character generator contains the colour of each dot by using two bits to represent each display dot. With a ROM character generator most characters will thus appear as an array of different coloured points rather than a character. See the section on "6561 Display Modes" for information on the use of user definable RAM character generators and high resolution point plotting.

In 'Multicolour' mode the two bits of the character generator character matrix which represent each screen dot select one of four colours for that dot. The four codes created by these two bits tell the 6561 where to find the colour information for the dot. The two bit code is not itself a colour code, it is simply a pointer to four different colour codes, giving greater flexibility, as each code pointed to has either 3 or 4 bit resolution. The use of a simple two bit pointer, combined with bit 3 of the ·colour video RAM being used to determine the colour display mode means that it is possible to freely intermix 'High resolution' and 'Multicolour' characters in a display. The colour of the dot can be either the background colour, the foreground colour, the exterior border colour or a special auxiliary colour (information on which is stored in bits 4-7 of control register No. 15.). The 'Multicolour' mode select codes are:

  0 0 — Background colour
  0 1 — Exterior border colour
  1 0 — Foreground colour
  1 1 — Auxiliary colour

The use of the 'Multicolour' mode can be summarised using the following example:

1 — Set the background colour to one of 16 colours, this colour code is stored in the following example in variable C which will have a value between 0 and 15:
```
POKE 36879,PEEK(36879)AND15
POKE 36879,PEEK(36879)OR(C*16)
```

2 — Set the exterior border colour to one of 8 colours, this colour code will have a value between 0 and 7 and in the following example is stored in variable C:

POKE 36879,PEEK(36879)AND248
POKE 36879,PEEK(36879)ORC

3 — Set the foreground colour to one of 8 colours by POKEing the colour code into the colour video RAM location, corresponding to the location of the displayed 'Multicolour' character. Since it is bit 3 of the colour video RAM which determines whether a character is displayed in 'High Resolution or Multicolour mode then 8 should be added to the colour code values for all characters to be displayed in 'Multicolour' mode.

4 — Set the auxiliary colour code to one of 16 colours, this colour code will have a value between 0 and 15 and in the following example is stored in variable C:
POKE 36878,PEEK(36878)AND15
POKE 36878,PEEK(36878)OR(C*16)
NOTE: Bit 3 of control register No. 16 has no function in 'Multicolour' mode but should be set to the normal value of 1, unless otherwise required when intermixing both colour display modes.

5 — Set up the character generator matrix for each character to be displayed, thus:

	bit								Hex	Location
byte	7	6	5	4	3	2	1	0		
0	0	0	0	1	1	0	1	1	1B	5120
1	0	0	0	1	1	0	1	1	1B	5121
2	0	0	0	1	1	0	1	1	1B	5122
3	0	0	0	1	1	0	1	1	1B	5123
4	0	0	0	0	0	0	0	0	00	5124
5	0	1	0	1	0	1	0	1	55	5125
6	1	0	1	0	1	0	1	0	AA	5126
7	1	1	1	1	1	1	1	1	FF	5127

This example is for a character in a user definable character generator starting at location 5120, the character has a code value of 0 and shows each of the four colours available in multicolour mode characters thus:

byte
0
1
2
3
4
5
6
7

7 6 5 4 3 2 1 0

Hex	Location
1B	5120
1B	5121
1B	5122
1B	5123
00	5124
55	5125
AA	5126
FF	5127

142

Sound Locations

The sound locations must be POKEd with numbers between 128 and 255. The frequency rises as the number, with the exception of 255 which is a low frequency. Each tone location produces one voice. A 0 in any byte will turn that voice off. The following decimal codes produce an approximation of three octaves of the even-tempered musical scale. The scale is relative, not absolute concert pitch. This table lists the musical note and its respective POKE location.

Poke locations of Musical Notes

Musical Note	Poke
C	128
C#	134
D	141
D#	147
E	153
F	159
F#	164
G	170
G#	174
A	179
A#	183
B	187
C	191
C#	195
D	198
D#	201
E	204
F	207
F#	210
G	213
G#	215
A	217
A#	219
B	221
C	223
C#	225
D	227
D#	228
E	230
F	231
F#	232
G	234
G#	235
A	236
A#	237
B	238
C	239
C#	240

143

THE 6561 SOUND GENERATORS

The sound effect or music generation capabilities of the VIC are controlled by five registers in the 6561. Four of the registers are associated with sound generation, the fifth controls the volume of the sound output. Each of the four sound generation registers has an associated oscillator, the register contents determine the frequency of the oscillator output. The frequency is determined by varying the pulse width, the output from all four oscillators is a symmetrical square wave. The outputs are combined to give the audio input to the TV display, where the sound is generated via the TV speaker. One of the four audio oscillators acts as a variable frequency noise source and the other three generate a simple tone. The five control registers used are:

Audio oscillator No. 1. — control register No. 11., location 36874. Bits 0-6 control the frequency, bit 7 turns the oscillator on (=1) or off (=0). Low base frequency, thus = value of 128 put in this register will produce the lowest frequency sound of any of the three audio oscillators.

Audio oscillator No. 2. — control register No. 12., location 36875. Bits 0-6 control the frequency, bit 7 turns the oscillator on (=1) or off (=0). The base frequency for this oscillator is between that for audio oscillators No. 1., and No. 3.

Audio oscillator No. 3. — control register No. 13., location 36876. Bits 0-6 control the frequency, bit 7 turns the oscillator on (=1) or off (=0). This has the highest base frequency of the three oscillators.

Noise generator — control register No. 14, location 36877. Bits 0-6 control the base frequency of the noise generator, bit 7 turns it on (=1) or off (=0). This is a pseudo white noise generator, giving a random sequence of pulses with a frequency determined by the contents of the control register.

Volume control — control register No. 15., location 36878. The volume of the composite audio signal produced when one or more of the four audio oscillators is turned on is controlled by bits 0-3.

Each of the audio oscillators is capable of generating 128 different frequencies and each oscillator is different, thus oscillator No. 1., can

be described as a 'base' sound generator, No. 2., as a 'Tenor' and No. 3., as a 'soprano'. The combined audio output has one of sixteen volume levels.

The four sound generators can be used to create a wide range of sound effects for use in games programs, they can also be used to play music. Writing routines to create sound effects is simply a matter of experimentation. Try to analyse the required sound and then re-create it using a combination of the four audio oscillators and the volume control, this is demonstrated in some of the following examples:

```
5 REM *MAKES A SOUND LIKE THE SINGING
6 REM *OF BIRDS
7 REM
10 POKE36878,15
20 FORL=1TO20
30 FORM=254TO240+INT(RND(1)*10)STEP-1
40 POKE36876,M
50 NEXTM
60 POKE36876,0
70 FORM=0TOINT(RND(1)*100)+120
80 NEXTM
90 NEXTL
100 GOTO10
```

```
5 REM*MAKES A SOUND LIKE THE RINGING OF
6 REM*A TELEPHONE
7 REM
10 POKE36878,15
20 FORL=1TO5
30 FORM=1TO50
40 POKE36876,230
50 FORN=1TO5
60 NEXTN
70 POKE36876,0
80 NEXTM
90 FORM=1TO3000
100 NEXTM
110 NEXTL
120 POKE36878,0
```

```
1 REM *MAKES A SOUND LIKE A GALLOPING
2 REM *HORSE, THE SOUND RECEDES INTO THE
3 REM *DISTANCE.
4 REM
5 FORX=15TO0STEP-1
6 FORZ=1TO4
7 A=60
10 POKE36878,X
20 POKE36876,230
30 POKE36876,0
40 FORQ=1TOA:NEXTQ
50 POKE36876,230
60 POKE36876,0
70 FORQ=1TOA:NEXTQ
110 POKE36878,INT(X/2)
120 POKE36876,230
130 POKE36876,0
140 FORQ=1TOA:NEXTQ
150 POKE36876,230
160 POKE36876,0
170 FORQ=1TO4*A:NEXTQ
180 NEXTZ
190 NEXTX
```

```
1 REM *MAKES A SOUND LIKE THE TICKING OF
2 REM *A GRANDFATHER CLOCK
3 REM
5 A=700
10 POKE36878,15
20 POKE36876,230
30 POKE36876,0
40 FORQ=1TOA:NEXTQ
50 POKE36876,236
60 POKE36876,0
70 FORQ=1TOA:NEXTQ
80 GOTO20
```

```
5 REM *MAKES A SOUND LIKE THE BREAKING
6 REM *OF WAVES ON A SEASHORE
7 REM
10 POKE36877,180
20 FORL=1TO10
30 D=INT(RND(1)*5)*50+50
40 FORM=3TO15
50 POKE36878,M
60 FORN=1TOD
70 NEXTN
80 NEXTM
90 FORM=15TO3STEP-1
100 POKE36878,M
110 FORN=1TOD
120 NEXTN
130 NEXTM
140 NEXTL
150 POKE36878,0
160 POKE36877,0
200 GOTO10
```

Using the audio generators on the 6561 to play music requires some thought otherwise the result will sound very abrasive and not very satisfactory. The first problem is that the square wave output from the audio oscillators produces a rather unpleasant set of harmonics which gives the note a rough sound. Only external electronics can change the shape of the waveform, but by using two audio oscillators to produce the same note of frequencies about an octave apart a more pleasing sound is produced. The second problem is to generate the correct attack and decay for the instrument, this is done by changing the amplitude of the output during the generation of each note. These two ideas are illustrated in the following program which plays scales, the sound resembles a piano.

```
1 REM *PLAYS A REPEATING OCTAVE SCALE
2 REM *THE SOUND OF EACH NOTE DECAYS AND
3 REM *THUS SOUNDS MORE LIKE A PIANO THAN
4 REM *AN ELECTRIC ORGAN
5 READA:IFA=100THEN150
10 POKE36874,A
20 POKE36875,A
30 FORQ=15TO0STEP-1
40 POKE36878,Q
50 FORX=1TO50:NEXTX
60 NEXTQ
100 GOTO5
150 RESTORE:GOTO5
200 DATA223,227,230,231,234,236,238,239
210 DATA239,238,236,234,231,230,227,223,100
```

To play a musical score requires a note table, this table contains each note in the score in the form of the value to be placed into the audio oscillator register and the duration of that note.
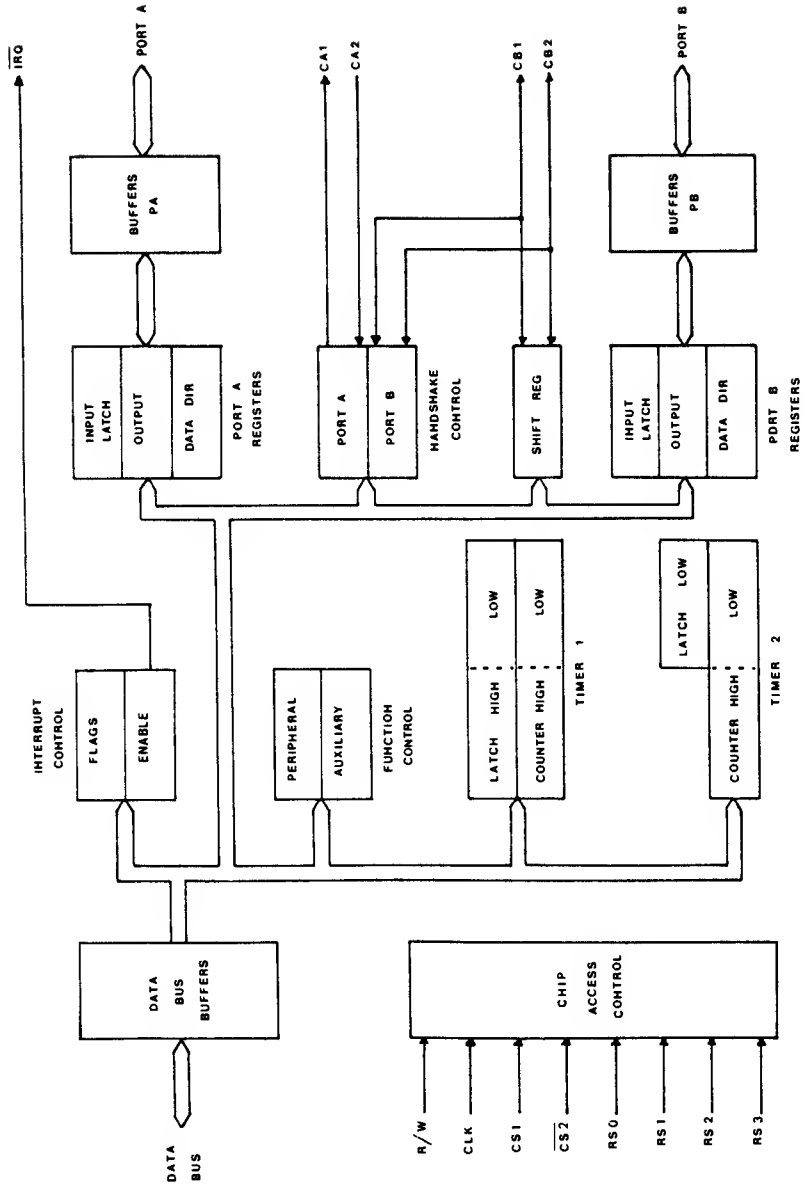
Fig. 18 — Block diagram of the 6522 Versatile Interface Adapter chip.

THE VIC I/O PORTS AND THE 6522.

The VIC communicates with the user, and with peripheral devices, via three integrated circuits. The most important of the three is the 6561 VIC chip, looked at in detail in the previous section. The chip controls the video display, sound generation plus the peripheral devices, a light pen and a joystick. The other two integrated circuits are 6522 Versatile Interface Adapters or VIAs and these are used to perform all other I/O functions of the VIC. We can summarise the function of these two chips as follows:

Keyboard input
User port
Cassette deck
Serial I/O — cut down IEEE 488 port
RS232 I/O — for printers modems etc.
Restore key (NMI line)
Joystick — simple switch type
Light pen control
IRQ timing for real time clock and keyboard

The two VIA chips which are used to control all these functions have between them just 32 programmable I/O lines and eight handshake lines. Many of these lines are used by more than one of the above functions.

An understanding of the two 6522 VIA interface chips is essential if all the features of the VIC are to be used to the full, and a knowledge of these chips helps to explain some of the quirks of the system. The functioning of these chips is controlled by internal programmable registers, there are sixteen registers in each chip. These 32 registers (sixteen from each chip) are located in addressable memory space and are located at hex $9110 - $912F (decimal 37136 to 37168). They can thus be accessed from Basic using PEEK and POKE statements and from machine code using LDA and STA commands.

Of the 40 I/O lines output from the two VIA chips, the user can directly connect equipment to, and control the functioning of, 23 lines, the other 17 lines are used by the keyboard and are not therefore usable. All but one of the I/O lines on VIA No. 1., can be used, but only five of the lines on VIA No. 2., VIA No. 1., is thus used in all the examples in this section. The functions of each I/O line from the two VIA chips is shown in Figure19, the electrical connections which allow the user to utilise some of these lines is shown in Figures 20 to 24
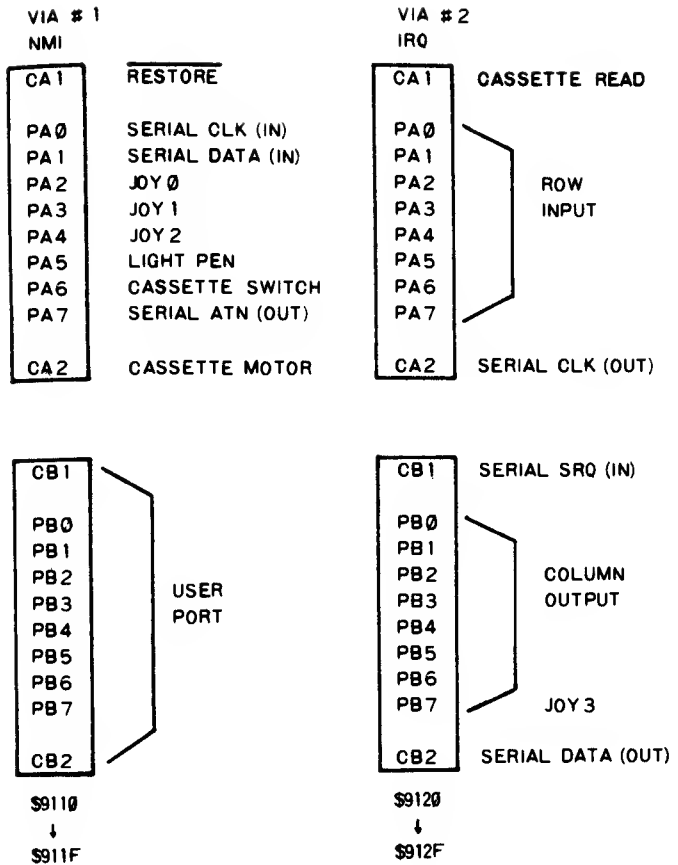
Fig. 19 — Allocation of the I/O lines from the two 6522 chips.

Though these lines are all assigned particular functions the user is not confined to using a particular I/O line for the function designated for that line. This is because all the I/O lines are under software control, and it is not until the routines, within the operating system which utilise that line, are called for a particular function, that that line is used. This flexability allows the re-definition of I/O line function and is one of the most useful features of the VIC.

A block diagram of the 6522 is shown in Figure 18, a very complex chip, with sixteen different addressable registers. Each bit within these registers has a specific function, either as an input, an output or to control the operation of the 6522. A memory map of the addressable registers of the two chips is shown in Figure 8, the registers are of six basic types; I/O data direction, peripheral control, shift register, timers and timer control registers.

The diagram in Figure 18 can be divided into two, on the left are the connections to the processor, the processor interface. On the right the outputs of the 6522, or the peripheral interface. The main components of the processor interface are the eight bi-directional data lines. These are connected directly to the processor data bus and are used to transfer data between the VIA and the processor. As with any memory, the processor treats the 6522 as a sixteen byte block of memory, the direction of data transfer is controlled by the R/W line, the exact timing of a transfer being controlled by the $\phi 2$ clock line. The individual registers are addressed by the register select lines connected to the bottom address lines AO — A3. The exact location of the 6522 within memory space is determined by de-coding some of the address lines and connecting these to the chip select inputs. The registers of the 6522 will only be accessed if chip select CS1 is high and CS2 low. As with all the I/O chips the 6522 can generate a processor interrupt by pulling the IRQ line low. This occurs whenever an internal interrupt flag is set as a result of an input on one of the peripheral control lines.

The processor interface lines have seven basic functions which can be summarised as follows:

1 — Phase Two Clock ($\phi 2$) — data transfers between the 6522 and the processor take place only when the $\phi 2$ clock is high. This clock also acts as a time base for the internal 6522 timers and shift register. On the VIC the $\phi 2$ clock is derived from the 6561 video interface chip and has a frequency of MHz 1.1082.
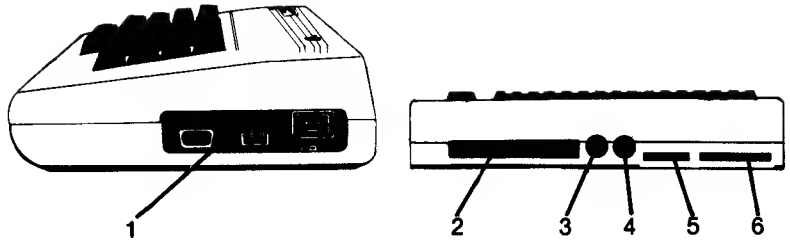
1) Game I/O   4) Serial I/O

2) Memory Expansion   5) Cassette

3) Audio and Video   6) User Port

Fig. 20 — Position of the different VIC I/O outputs.

USER I/O
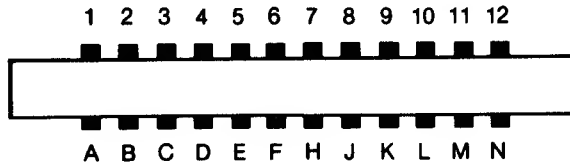
PIN #	TYPE	NOTE	PIN #	TYPE	NOTE
1	GND		A	GND	
2	+5V	100mA MAX.	B	CB1	
3	RESET		C	PB0	
4	JOY0		D	PB1	
5	JOY1		E	PB2	
6	JOY2		F	PB3	
7	LIGHT PEN		H	PB4	
8	CASSETTE SWITCH		J	PB5	
9	SERIAL ATN IN		K	PB6	
10	+9V	100mA MAX.	L	PB7	
11	GND		M	CB2	
12	GND		N	GND	

Fig. 21 — The allocation and function of pins on the User Port connector.

155

2 — Chips Select Lines (CS1, CS2) — the two chip select inputs are connected to the processor address bus. CS1 is connected directly to a low address line, in VIA No. 1., to A4 and in VIA No. 2., to A5. CS2 is connected in both VIA chips to a decoded address derived from address lines A10 to A15. CS2 determines the starting address as $9100, and CS1 is the offset so that VIA No. 1., starts at address $9110 and VIA No. 2., at $9120. Note that the 6522 registers can only be addressed when CS1 is high and CS2 is low.

3 — Register Select Lines (RS0, RS1, RS2, RS3) — the four register select lines are connected to the processor address bus lines A0 — A3. This allows the register to select one of the sixteen registers in the 6522.

4 — Read/Write Line (R/W) — the direction of data transfer between the 6522 and the processor is controlled by the R/W line. If R/W is high then a 'read' operation is performed and data is transferred from the 6522 onto the data bus. If R/W is low then a 'write' operation is performed and data currently on the data bus is loaded into the addressed register of the 6522.

5 — Data Bus (DB0 to DB7) — data is transferred between the processor and the 6522 via the eight bi-directional lines of the data bus. The internal data bus of the 6522 will only be connected to the processor data bus when the two chip select lines are enabled and the $\phi2$ clock is high. The direction of data transfer will depend on the state of the R/W line and the register addressed on lines RS0 to RS3.

6 — Reset (RES) — the reset line clears all the internal registers of the 6522 (except the timers and shift register) and sets them all at logic zero. Resulting in all the interface lines put in the input state, and timers shift registers and interrupts are all disabled. This is connected to the processor power up circuitry and is only used when the system is switched on (this line is accessable externally and since the system software can be changed its function could be modified).

7 — Interrupt Request (IRQ) — the interrupt request output from the 6522 is very important in the VIC. The IRQ line goes low whenever an internal interrupt flag is set and the corresponding interrupt flag is high. On VIA No. 1., the IRQ line is connected to the processor NMI interrupt line, this is used to test the RESTORE key which is connected to the CA1 line of the VIA, an IRQ signal is produced if this line is

SERIAL I/O
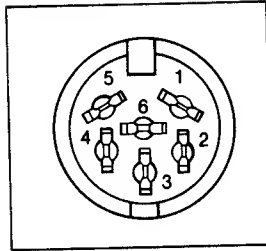
PIN #	TYPE
1	SERIAL SRQ IN
2	GND
3	SERIAL ATN IN/OUT
4	SERIAL CLK IN/OUT
5	SERIAL DATA IN/OUT
6	NC

Fig. 22 — The allocation and function of pins on the Serial I/O connector.

AUDIO/VIDEO
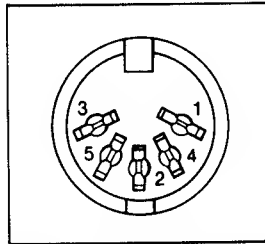
PIN #	TYPE	NOTE
1	+6V	10mA MAX
2	GND	
3	AUDIO	
4	VIDEO LOW	
5	VIDEO HIGH	

Fig. 23 — The allocation and function of pins on the Audio/Video connector.

GAME I/O
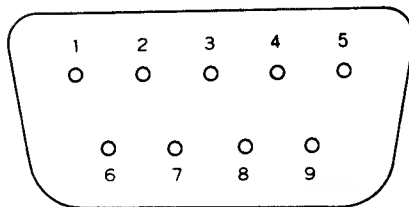
PIN #	TYPE	NOTE
1	JOY∅	
2	JOY1	
3	JOY2	
4	JOY3	
5	POT Y	
6	LIGHT PEN	
7	+5V	MAX. 100mA
8	GND	
9	POT X	

Fig. 24 — The allocation and function of pins on the Game I/O connector.

157

brought low (key depressed). On VIA No. 2., the IRQ line is connected to the processor IRQ line, the function of this line is to generate a regular 60 Hz interrupt which is used by the clock, I/O and keyboard routines, this interrupt is provided by Timer 1 in the VIA.

The peripheral interface lines are divided into two I/O ports, each port having eight bi-directional I/O lines and two control lines. The function of each of the four ports on the VIC are shown in Figure 3. The following is a brief discription of the I/O buses and control lines of a 6522:

1 — Peripheral A Port (PA0 — PA7) — this port consists of eight bi-directional lines each of which can be independently programmed under control of the Data Direction Register to act as either an input or an output. The polarity of the lines defined as outputs is controlled by the contents of the Output Register. Data input on those lines defined as inputs can be latched into an internal register under control of the CA1 line. The internal control registers are used by the processor to control the modes of operation of the 6522. All lines represent. a load of one standard TTL gate in the input mode and will drive one standard TTL load in the output mode.

2 — Peripheral A Control Lines (CA1, CA2) — the two peripheral control lines act as interrupt inputs (as in the RESTORE key) or as handshake outputs (as in serial clock output). Each line controls an internal interrupt flag with a corresponding interrupt enable bit. In addition CA1 controls the latching of data on peripheral port A input lines. The various modes of operation are control registers of the 6522. CA1 is a high impedence input only while CA2 is either an input or an output. In the input mode CA2 will source one standard TTL load and in the output mode will drive one standard TTL load.

3 — Peripheral B Port (PB0 — PB7) — this port consists of eight bi-directional lines each of which can be independently programmed under control of a Data Direction Register to act as either an input or an output. The operation and electrical characteristics of Port B is the same as Port A. In addition when line PB7 is in the output mode it's polarity can be controlled by one of the internal timers. The second timer can be used to count pulses on line PB6 when that line is in the input mode.

4 — Peripheral B Control Lines (CB1, CB2) — these two peripheral control lines have the same functions and electrical characteristics as

control lines CA1 and CA2. They also have the additional function of acting as a serial port under control of the internal shift register of the 6522.

OPERATION OF THE I/O PORTS

Three registers are required to access each of the eight line peripheral ports, they are a Data Direction register, an Output register and an Input register. Each port has a Data Direction register for specifying whether each of the eight lines acts as either an input or an output. A zero in bit of the Data Direction register causes the corresponding peripheral line to act as an input. A one causes the line to act as an output.

Example: Set lines 0 to 3 as inputs and 4 to 7 as outputs on Port B of VIA No. 1.

I/O line number	data direction	DDR contents if line = in	DDR contents for example
0	in	1	0
1	in	2	0
2	in	4	0
3	in	8	0
4	out	16	16
5	out	32	32
6	ou	64	64
7	out	128	128

Total for example — 240

Command is — POKE 37138,240

Each peripheral line is connected to an Input register and an Output register. When a line is programmed to act as an output the voltage on that line is controlled by the corresponding bit in the Output register. A '1' in the Output register causes the corresponding line to go 'high', and a '0' causes it to go 'low'.

Example: output to port B of VIA No. 1., using the data direction set out in the previous example, lines 4 and 7 are high and lines 5 and 6 are low.

I/O line number	data direction of line	'high' or 'low'	value of line in I/O reg
0	in	—	—
1	in	—	—
2	in	—	—
3	in	—	—
4	out	'high'	16

5	out	'low'	0
6	out	'low'	0
7	out	'high'	128

Total for example — 144

Command is — POKE 37136,144

Reading one of the peripheral port registers causes the contents of the Input register to be transferred onto the data bus. With input latching disabled the contents of the Input registers will always reflect the data currently on all the peripheral port lines.

Examples: read the contents of the input lines of port B of VIA No.1., set up in the example on data direction, store the contents as variable A.

A = PEEK (37136) AND 15

the AND 15 masks off the lines used as outputs, they must be removed since the current state of the output lines is stored in the input register. AND commands can then be used to determine which lines are 'high' and which are 'low'; thus to determine the state of line use:

```
10 X = A AND 2: REM line 2 = AND 4, line 3 = AND 8 etc
20 IF X = 0 THEN 40
30 PRINT "Line 1  is 'high' :END
40 PRINT "Line 1 is 'low":END
```

If input latching is enabled then input register A will reflect the contents of all the lines on Port A prior to the setting of the CA1 interrupt flag by an active transition on CA1.

There is a slight difference in the operating of Port A and Port B. Both operate in the same manner, however, for the output lines of Port B the corresponding bits in the Input register will contain the contents of the corresponding output register bit instead of the data on that line. This allows proper data to be read into the processor if the output line is not allowed to go to the full voltage. Thus if an output line is tied to ground or zero voltage then that line will always be at a logic zero irrespective of the contents of the corresponding Output register bit. In Port A this bit will always be '0' in the Input

161

register but in Port B it will contain the contents of the corresponding bit in the Output register. With input latching enabled on Port B, setting the CB1 interrupt enable flag will cause the Input register to latch this combination of input data and Output register data until the interrupt flag is cleared.

Registers Used in Operation of the I/O Ports.

Register 1 — Parallel port B I/O register
VIA No. 1 — Hex $9110 decimal 37136
VIA No. 2 — Hex $9120   decimal 37152

This register contains the contents of the input and output lines of port B of the 6522, Reading

CB2 interrupt flag to be reset.

Register 2 — Parallel port B I/O register with handshake control.
VIA No. 1 — Hex $9111   decimal 37137
VIA No. 2 — Hex $9121   decimal 37153

This is one of two registers which contain the contents of the input and output lines of port A. The two registers are identical except that this register has control over the handshake lines. Data input using the CA1 line to latch the input into the I/O register will set the CA1 interrupt flag. This flag is cleared by reading the contents of register No. 2.

Register 3 — Data direction register for port B
VIA No. 1 — Hex $9112   decimal 37138
VIA No. 2 — Hex $9122   decimal 37154

This register controls each of the eight lines on Port B and determines whether they are acting as inputs or outputs. A 'one' in any of the eight bits of this register sets the corresponding line into the output mode, and a 'zero' puts it into the input mode.

Register 4 — Data direction register for port A
VIA No. 1 — Hex $9113   decimal
VIA No. 2 — Hex $9123   decimal 37155

This register controls each of the eight lines on port A and determines whether they are acting as inputs or as outputs. A one in

any of the eight bits of this register sets the corresponding line into the output mode and a zero puts it into the input mode.

Register 16 — Parallel port A I/O register
VIA No. 1 — Hex $911F   decimal 37151
VIA No. 2 — Hex $912F   decimal 37167

This is the second of the two registers containing the contents of the input and output lines or port A. This register has no control over the handshaking lines. The direction of the data transfer in this port is controlled as in the other port A register by the contents of Data direction register A.

THE INTERVAL TIMERS AND COUNTERS OF THE 6522

The 6522 has two internal timers, one of which can also function as a counter of pulses input on one of the I/O lines. These timers are not only useful but of vital importance to the operation of the VIC. It is these timers which are used to control the generation of the 60 Hz interrupt used to update the real time clock and scan the keyboard. They are also used to control the timing of I/O on the serial port, the RS232 port and the cassette. Since the VIC interface uses two 6522 chips there are a total of four timers available for use by the system software. The timers are used in conjunction with the processor interrupts, the following table shows some of the functions of each timer plus the interrupt line affected:

VIA No. 1   NMI interrupt

Timer 1 — RS232 port I/O timing
            user port operation

Timer 2 — RS232 port I/O timing
            user port operation

VIA No. 2   IRQ interrupt

Timer 1 — System 60 Hz interrupt
            real time clock updating
            keyboard scanning
        Cassette read/write timing

Timer 2 — Cassette read/write timing
            Serial port timing

Note that Timer 1 of VIA No. 2., is used for both updating the real time clock and cassette timing, for this reason the real time clock looses whenever the cassette is used.

Interval Timer 1 consists of two eight bit latches and a sixteen bit counter, these occupy four of the 6522 registers, registers number 5 to 18. Their location in the VIC are as follows:

Register 5 — Timer 1 counter low order byte
            VIA No. 1 — Hex $9114   decimal
            VIA No. 2 — Hex $9124   decimal

Register 6 — Timer 1 counter high order byte
          VIA No. 1 — Hex $9115   decimal 37141
          VIA No. 2 — Hex $9125   decimal 37157

Register 7 — Timer 1 latch low order byte
          VIA No. 1 — Hex $9116   decimal 37142
          VIA No. 2 — Hex $9126   decimal 37158

Register 8 — Timer 1 latch high order byte
          VIA No. 1 — Hex $9117   decimal 37143
          VIA No. 2 — Hex £9127   decimal 37159

The latches are used to store data to be loaded into the counter. After loading, the counter decrements at the system clock rate (MHz 1.1082). Thus if the counter is loaded with its maximum value (all sixteen bits = or decimal 65,535) it will be decremented to zero in .0591 seconds. Upon reaching zero, an interrupt flag is set and one of the two interrupt lines will go low and generate a processor interrupt. The timer will then disable any further interrupts, or automatically transfer the contents of the latches into the counter and continue to decrement. In addition the timer can be instructed to invert the output level on one of the peripheral I/O lines each time it 'times out'. The modes of operation are controlled by reading or writing to the four timer registers, plus the auxiliary control register and the two interrupt registers.

The processor can only load data directly into three of the four registers of Timer 1, these registers are the high order and low order, latch registers and the high order counter register. The low order counter register can only be loaded by an automatic process, which takes the contents of the low order latch and stores it in the low order counter register when the processor writes to the high order counter register. The following commands would have these effects on Timer 1 of VIA No. 1:

    POKE 37140, 255   :put 255 into the low order latch.
    POKE 37141, 255   :put 255 into the high order latch.
                        then transfer to high order counter then transfer
                        low order latch into low order counter,
                        and reset the Timer 1 interrupt flag.
    POKE 37142, 255   :put 255 into the order latch
    POKE 37143, 255   :put 255 into the high order latch and reset
                          the Timer 1 interrupt flag

All four Timer 1 registers can be read using PEEK or similar commands, reading each of these registers of VIA No.1., has the following effects:

A = PEEK(37140) : read contents of low order and reset Timer 1 interrupt flag.

A = PEEK(37141) : read contents of high order counter

A = PEEK(37142) ; read contents of low order latch

A = PEEK(37143) : read contents of high order latch

The four operating modes of Timer 1 are controlled by two bits in the Auxiliary Control Register. This register is register No. 12., of the 6522 and is located at address $911B (decimal 37147) for VIA No. 1., and $912b (decimal 37163) for VIA No. 2. Bits 6 and 7 of the Auxiliary Control register (for ACR) are used to control the operating modes of Timer 1, these four modes can be divided into two groups each of two modes, the one shot modes and the free running modes. The Auxiliary Control Register is also used to control the output by Timer 1 of pulses on peripheral I/O line PB7 of the VIA. However, to output pulses on PB7 requires that the Data Direction Register for Port B line 7 is set to '1' so that this line is in the output mode. Setting bit 7 of the ACR will then ensure that peripheral line PB7 is under control of Timer 1. Line PB7 is set into the output mode by the following command.

POKE 37138, PEEK (37138) AND 127 OR 128

Bits 6 and 7 of the ACR are used to control the operating modes as follows:

ACR bit 6 — '0' = enable one shot mode
            '1' = enable free-running mode

ACR bit 7 — '0' = disable output on PB7
            '1' = enable output on PB7

The four operating modes formed by combinations of these two bits can be obtained by using PEEK and POKE commands plus logical operators, thus for VIA No. 1., the modes and commands are:

ACR6 = '0'   ACR7 = '0'

Mode function — Generate a single time-out interrupt on (NMI) each time Timer 1 is loaded. Output of PB7 by Timer 1 is disabled.

POKE 37147, PEEK(37147) AND 63
ACR6 = '0'   ACR7 = '1'

Mode function — Generate continuous interrupts, the spacing between interrupts being determined by the contents of Timer 1. Output on PB7 by Timer 1 is disabled.

POKE 37147, PEEK(37147) AND 63 OR 64

ACR6 = '1'   ACR7 = '0'

Mode function — Generate a single interrupt and an output pulse on PB7 for each Timer 1 load operation. Note: ensure that the Data Direction Register is set to allow PB7 to function as an output before using this mode.

POKE 37147, PEEK(37147) AND 63 OR 128

ACR6 = '1'   ACR7 = '1'

Mode function — Generate continuous interrupts and pulses on PB7, the spacing between interrupts and pulses being determined by the contents of Timer 1. Note: ensure that the Data Direction Register is set to allow PB7 to function as an output before using this mode.

POKE 37147, PEEK(37147) AND 63 OR 192

The one-shot mode allows the generation of a single interrupt for each timer load operation. The sequence of events is that the timer is loaded with a value, the counter then decrements this value, when zero is reached an interrupt is generated. The delay between the write operation and the generation of the interrupt is thus directly proportional to the data loaded into the counter. If the operating mode and Data Direction Register contents allow the generation of a pulse on peripheral line PB7 then the pulse width is also dependent on the value loaded into the counter. To use the timer in the one-shot mode the following sequence of operations are performed:

1 — Set bits 6 and 7 of the Auxiliary Control Register to give the correct operating mode — one-shot with output on PB7 and without output on PB7. If a pulse is to be output on PB7 ensure that the Data Direction Register is correctly set to define PB7 as an output.

167

2 — Load the low order latch (location $9116 of VIA No. 1) with low order part of value to be loaded into the counter.

3 — Load the high order counter (location $9115 of VIA N0. 1) with the high order part of the timing value.

Operation number 3 initiates the following events:

1 — The contents of the low order latch are transferred into the low order counter

2 — If the PB7 output is enabled then this line will be pulled low on the phase two clock pulse following the write to high order counter.

3 — The contents of the counter is decremented at the $\phi2$ system clock rate.

4 — When the counter reaches zero the Timer 1 interrupt flag is set and a system interrupt generated (assuming the interrupt is enabled), if the output on PB7 is enabled then that line will go high.

5 — The counter will roll over and continue decrementing from a value of decimal 65,535. This allows the system processor to read the counter contents and determine the time since the interrupt. The Timer 1 interrupt flag will not be reset until it has been cleared by a read of low order counter or a write to high order latch).

Note: when using the timers, the count value loaded into the timer must be two counts less than the desired interval time, this is due to the 1 1/2 cycle overhead on interval timing.

The free-running mode allows the generation of a continuous series of evenly spaced interrupts. If the operation mode and Data Direction Register contents allow the generation of an output on peripheral line PB7 then a continuous series of pulses are also produced on this line. The time between each interrupt or output pulse is dependent on the contents of the Timer 1 latch bytes. To use the timer in the free-running mode the following sequence of operations are performed:

1 — Set bits 6 and 7 of the Auxiliary Control Register to give the correct operating mode — free-running with output on PB7 and without output on PB7. If pulses are to be output on PB7 ensure that

the Data Direction Register is correctly set to define PB7 as an output.

2 — Load the low order latch (location $9116 of VIA No. 1) with the least significant byte of the counter delay value).

3 - Load the high order latch (location $9115 of VIA No. 1) with the most significant part of the counter delay value.

This will initiate the following sequence of events:

1 — The counter of Timer 1 will be loaded with the contents of the two latch registers.

2 — The counter will start decrementing at the $\phi 2$ system clock rate.

3 — When the counter reaches zero the interrupt flag will be set and if Timer 1 interrupts are enabled a system interrupt will be generated. The output on line PB7 will be inverted (it will go low on the first interval) if outputs on this line are enabled.

4 — The latch contents will be reloaded into the counter and the process repeated through from step 1 to 4 until disabled by changing the operating mode.

Timer 2 occupies two registers of the 6522, one contains the low order latch and counter value, the second contains the high order counter, together they comprise a single sixteen bit counter. The location of these two registers, number 9 and 10, in the VIC are as follows:

Register 9 — Timer 2 low order latch/counter
         VIA No. 1. — Hex $9118   decimal 37144
         VIA No. 2. — Hex $9128   decimal 37160

Register 10 — Timer 2 high order counter
         VIA No. 1 — Hex $9119   decimal 37145
         VIA No. 2 — Hex $9129   decimal 37161

The operation of register 9 is a 'write only' latch and a 'read only' counter which contains the least significant byte of the the counter value. The functions of the two registers are summarised in the following commands to the registers of Timer 2 of VIA No. 1.

169

POKE 37144, 255   :put 255 into the low order latch

A = PEEK (37144)   :variable A will contain the value in the low order counter. This will clear the Timer 2 interrupt flag.

POKE 37145, 255   :put 255 into the high order counter and transfer the contents of the low order latch to the low order counter. Will clear the Timer 2 interrupt flag.

A = PEEK(37145)   A = contents of high order counter

Timer 2 has two modes of operation, these modes are selected by the contents of bit 5 of the Auxiliary Control Register. In the first mode Timer 2 acts as an internal timer (in the one-shot mode only) and in the second mode as a counter of pulses input on peripheral line PB6. The ACR contents controlling these two modes are as follows:

ACR bit 5 — '0' = one-shot interval timer
'1' = pulse counting mode

The mode can be set with the following Basic commands:

ACR5 = '0'   POKE 37147, PEEK(37147) AND 223

ACR5 = '1'   POKE 37147, PEEK(37147) AND 223 OR 32

The operation of Timer 2 in the one-shot interval timer mode is similar to the same mode on Timer 1 allowing the generation of a single interrupt for each timer load operation. The delay between the write operation and the interrupt is proportional to the value stored in the two counter registers, this value being decremented at the $\phi 2$ system clock rate. When the value reaches zero the interrupt is generated. To use the timer in this mode the following sequence of operations is performed:

1 - Set bit 5 of the ACR to logic '0' to give the correct operating mode.

2 — Write the least significant byte of the interval count value into the low order latch (location $9118 of VIA No. 1).

3 — Write the most significant byte of the interval count value into the high order counter (location $9119 of VIA No. 1).

Operation 3 initiates the following events:

1 — The contents of the low order latch are transferred into the low order counter.

2 — The contents of the counter have decremented at the $\phi 2$ system clock rate.

3 — When the counter reaches zero the Timer 2 interrupt flag is set and if the interrupt is enabled a system interrupt generated.

4 — The counter will roll over and continue to decrement from a value of decimal 65,535. This allows the system processor to determine the time since the interrupt. The Timer 2 interrupt flag will not be reset until it is cleared by reading the contents of the low order latch or writing to the high order counter. The latter will also initiate a new interval timing sequence.

In the pulse counting mode the function of Timer 2 is to count a predetermined number of negative going pulses input on peripheral line PB6, the counter can also be used as a straight pulse counter. The two registers of Timer 2 are loaded with a value. Writing into the high order byte initiates the countdown and clears the interrupt flag. The counter contents are decremented each time a pulse is applied to line PB6. When the counter reaches zero the interrupt flag is set. To use Timer 2 in the pulse counting mode requires the following sequence of operations:

1 — Set bit 5 of the ACR to logic '1' to give the correct operating mode.

2 — Set the Data Direction Register so that line PB6 is an input, this can be done with the following command (the register location is for VIA No. 1).

POKE 37138, PEEK(37138) AND 191

3 — Load the low order latch (location $9118 of VIA No. 1) with the least significant byte of pulse count value.

171

4 — Load the high order counter (location $9119 of VIA No. 1) with the most significant byte of the pulse count value.

Operation 4 initiates the following sequence of events:

1 — The contents of the low order latch is transferred into the low order counter and the interrupt flag is cleared.

2 — The counter contents are decremented on a negative going pulse. This pulse must have gone low prior to the leading edge of the $\phi2$ clock pulse, if not then the counter will not be decremented until the next $\phi2$ clock pulse.

3 — The counter continues to be decremented by pulses input on PB6 until the counter reaches zero. On reaching zero the interrupt flag is set.

4 — The counter will roll over and contain the value decimal 65,535, further pulses input on PB6 will continue to decrement this value. However, it will be necessary to rewrite the contents of the high order counter or read the low order counter to reset the interrupt flag.

Timer 2 is also used to control the frequency of input or output from the internal shift register of the 6522.

Both the internal timers on the 6522 have several features in common, the principle being that they are all 'retriggerable'. This means that the time-out period will always be re-initialised by rewriting the counter. The value of this is that a time-out and its associated interrupt can be prevented if the processor rewrites the timer prior to its reaching zero. This is utilised by all the time-out features of the I/O software on the VIC, thus allowing proper detection of time-out errors. The second important feature is that both counters have a 1 1/2 clock cycle 'overhead' on the time interval. This means that the count value loaded into the timer must have a value, two counts less, than the required interval.

172

THE SHIFT REGISTER

One of the internal registers of the 6522, register 11, functions as a shift register, converting data between serial and parallel format. Eight data lines, one control line and a ground line, are required to transfer a byte of data from a peripheral I/O port like a VIA to an external device. With serial transmission this same byte of data can be transferred using just three lines, a data line, a control line and a ground line. Serial data transmission is thus of considerable use in controlling and communicating with external devices, where the number of lines connecting the computer to the external device must be kept to a minimum. The internal shift register of the 6522 allows this serial input/output of data, though its functioning and flexibility is less sophisticated than a standard UART (Universal Asynchronous Receiver Transmitter). Besides generating or inputting serial data, the shift register can perform a range of other functions, including variable frequency pulses output on one of the VIA I/O lines and a means of expanding the I/O capability of the VIA.

Two I/O lines are associated with the operation of the shift register, they are CB1 and CB2. The CB2 line is used for the serial transmission of the byte of data either into or out of the shift register. CB1 is used to carry the internally or externally generated shift clock pulses which clock the serial data in or out of the shift register. Each pulse on the shift clock shifts the entire contents of the shift register one bit to the left, bit 7 being output on line CB2 or the current logical state of the CB2 line input into bit 0. The contents of the shift register can be read or data loaded into it, by the standard processor read or write commands. There are three sources of the shift clock each having a different function and application, they are:

1 — Timer 2 low order register. The bottom eight bits of the 16 bit Timer 2 counter are used to create a programmable rate shift clock. The value loaded into the least significant byte of the timer controls the time between transitions of the shift clock, two transitions are required for one complete shift clock cycle. The value loaded into the timer can be between 0 and 255 and the timer count down rate is determined by the frequency of the processor $\phi 2$ clock, in the VIC this is MHz 1.1082. A delay time between transitions of 4.433 to 571.831 microseconds can be programmed between each shift operation. The shift clock pulses generated by Timer 2 are output on line CB1.

173

2 — Microprocessor $\phi2$ clock. The processor clock can be programmed to directly provide the shift clock pulses. The $\phi2$ clock signal is divided by two, this gives a maximum shift clock on the VIC of KHz 554.1 which gives a delay between each shift of 1.8 microseconds.

3 — An external signal applied to the CB1 line. This square wave signal input on CB1 can be any frequency subject only to a maximum of KHz 554.1.

The shift clock pulses are counted by a modulo-8 counter. When this counter has counted eight shift pulses it sets the shift register flag in the interrupt flag register. When the processor reads or writes data to the shift register the shift register flag is cleared. By clearing the shift register flag the modulo-8 counter is set to zero, the shift clock enabled, and data shifted in or out of the shift register. After eight shift pulses the flag is set and the shift clock disabled. In some modes of operation the modulo-8 counter is re-triggered by a write command to the shift register during a shift operation. In the free-running mode the modulo-8 counter is not used, resulting in continuous repeated output of the contents of the shift register.

The shift register of the 6522 has eight modes of operation and the mode used is selected by setting bits 2, 3 and 4 of the Auxiliary Control Register. The eight operating modes can be divided into four output modes and four input modes, for VIA No.1., the modes and commands are:

ACR2 = '0'   ACR3 = '0'   ACR4 = '0'

Mode function — The shift register is disabled.

POKE 37147, PEEK(37147) AND 227

ACR2 = '1'   ACR3 = '0'   ACR4 = '0'

Mode function — Input data on line CB2 and put into bit 0 of shift register, under control of Timer 2 low order counter, shift clock pulses output on CB1. Note: does not appear to work properly under Basic commands.

POKE 37147, PEEK(37147) AND 227 OR 4

ACR2 = '0'   ACR3 = '1'   ACR4 = '0'

Mode function — Input data on line CB2 under control of the system ɸ2 clock and put into bit 0 of shift register, shift clock pulses output on CB1. Note: does not appear to work properly under Basic commands.

POKE 37147, PEEK(37147) AND 227 OR 8

ACR2 ='1'   ACR3 = '1'   ACR4 = '0'

Mode function — Free running output of the shift register contents under control of Timer 2. In this mode the contents of the shift register are recirculated, bit 7 of the shift register is shifted onto the CB2 output line and simultaneously shifted into bit 0 of the shift register. The frequency of the shift pulses is determined by the contents of the low order byte of Timer 2. The result is the continuous repeated output of the contents of the shift register. In this mode the CB2 line can be used as a programmable frequency source or a simple music generator (by connecting CB2 to an amplifier and speaker).

POKE 37147, PEEK(37147) AND 227 OR 16

ACR2 = '1'   ACR3 = '0'   ACR4 = '1'

Mode function — Output data from shift register on line CB2 under control of Timer 2. The time delay between shift pulses is determined by the contents of the low order byte of Timer 2. A PEEK command to the shift register will reset the shift register flag in this mode even though the shift process is not completed.

POKE 37147, PEEK(37147) AND 227 OR 20

ACR2 = '0'   ACR3 = '1'   ACR4 = '1'

Mode function — Output data from shift register on line CB2 under control of the system ɸ2 clock.

POKE 37147, PEEK(37147) AND 227 OR 24

ACR2 = '1'   ACR3 = '1'   ACR4 = '1'

Mode function — Output data from shift register on line CB2 under control of an external shift clock input on line CB1.

POKE 37147, PEEK(37147) AND 227 OR 28

INTERRUPTS

The 6522 VIA has only a single interrupt request line to the 6502 microprocessor, VIA No. 1., interrupt is connected to the processor NMI interrupt and VIA No. 2., is connected to the IRQ interrupt. The interrupt output from the VIA can be activated (pulled to a logic '0' level) by any one of seven different conditions. These conditions are represented by bits in the Interrupt Flag Register (IFR), each bit or 'flag' can be set to either logic '1', the 'on' state, or to logic '0' the 'off' state. These flags are set as a result of certain conditions arising from the use of the other registers of the VIA. However, for one of these flags to activate the IRQ line to the processor requires two things to be true:

1 — The bit in the IFR that represents the condition which generates the interrupts must be 'on' and thus set to logic '1'.

2 — The corresponding bit in the Interrupt Enable Register (IER) must also be set to a logic '1'.

Without the correct bits in the IER being set an interrupt condition will only set a flag in the IFR and will not generate a processor interrupt. These two registers, the Interrupt Flag Register and the Interrupt Enable Register are thus directly connected in their function. The location of these two registers, number 14 and 15 in the VIC are as follows:

Register 14 — Interrupt flag register
        VIA No. 1 — Hex $911D   decimal 37149
        VIA No. 2 — Hex $912D   decimal 37165

```
7 6 5 4 3 2 1 0
│ │ │ │ │ │ │ └── CA2 interrupts flag
│ │ │ │ │ │ └──── CA1 interrupt flag
│ │ │ │ │ └────── Shift Register interrupt flag
│ │ │ │ └──────── CB2 interrupt flag
│ │ │ └────────── CB1 interrupt flag
│ │ └──────────── Timer 2 interrupt flag
│ └────────────── Timer 1 interrupt flag
└──────────────── IRQ has occurred
```
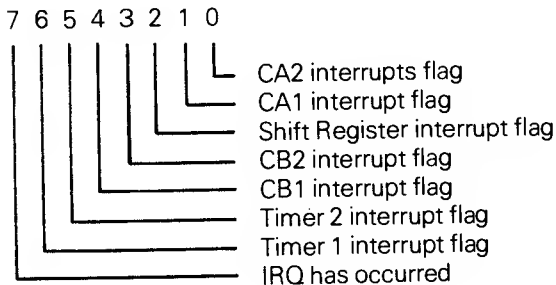
Register 15 — Interrupt enable register
                VIA No. 1 — Hex $911E   decimal 37150
                VIA No. 2 — Hex $912E   decimal 37166

```
7 6 5 4 3 2 1 0
            └─── CA2 interrupt enable
          └───── CA1 interrupt enable
        └─────── Shift register interrupt enable
      └───────── CB2 interrupt enable
    └─────────── CB1 interrupt enable
  └───────────── Timer 2 interrupt enable
 └────────────── Timer 1 interrupt enable
└─────────────── IER set/clear control
```
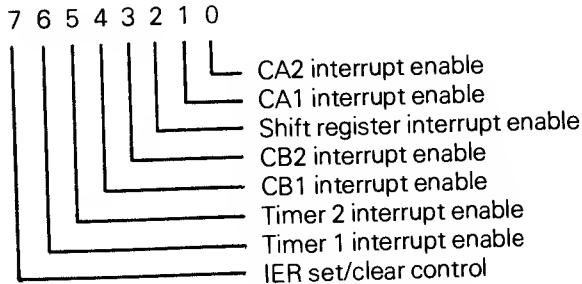
Each flag in the Interrupt Flag Register is set or cleared by a specific operation to one or more of the other registers of the 6522, this is summarised in the following table:

BIT	SET BY	CLEARED BY
0	Active transition on CA2	Reading or writing Output Register A
1	Active transition on CA1	Reading or writing Output Register A
2	Completion of 8 shifts	Reading or writing to Shift Register
3	Active transition on CB2	Reading or writing Output Register B
4	Active transition on CB1	Reading or writing Output Register B
5	Timeout on Timer 2	Reading Timer 2 counter low byte or writing Timer 2 high byte
6	Timeout of Timer 1	Reading Timer 1 counter low byte or writing Timer 1 high byte
7	Any IFR bit with corresponding IER bit also set	Writing logic '0' to the appropriate bit in IFR or IER

Bit 7 of the Interrupt Flag Register is connected to the IRQ output line of the VIA , and is set to logic '1' when one of the other seven

lower order bits in the IFR is set, it thus signals the condition that an interrupt has occured. Bit 7 of the Interrupt Enable Register is used to control the contents of the other seven lower order bits. If bit 7 of the IER is set to logic '1' then all other bits in that register which are set to logic '1' will enable the interrupt request corresponding to that bit. Thus the Timer 1 interrupt to the processor would only be enabled if bits 7 and 6 are both set to logic '1'. If bit 7 of the IER is set to logic '0' then any of the seven lower order bits of the IER which are set to logic '1' will disable the interrupt request corresponding to that bit. Thus all processor interrupts from a VIA could be disabled by setting bit 7 of the IEP to '0' and all other seven bits to '1'. Initialising the IER takes two write operations, one to select the enabled interrupt conditions and the other to select the disabled conditions. It should be noted that bit 7 of the IER is only active during a Write operation, when the contents are read it will always contain a logic '1' irrespective of its actual contents. The enabling or disabling of interrupts using the IER does not affect the setting of interrupt flags in the IFR.

The NMI interrupt is connected to the IRQ output of VIA No. 1., and the IRQ interrupt is connected to the IRQ output of VIA No. 2. When one or other of these interrupt lines is pulled low by an interrupt from one of the VIA's, the processor completes the current instruction pushes the register contents and program counter onto the stack and jumps to either the NMI of IRQ interrupt handling routines. The starting addresses of the two interrupt handling routines are stored in the top few bytes of memory. The NMI start address is located at $FFFA and $FFFB and the IRQ start address at $FFFE and $FFFF. In the VIC the start addresses of the interrupt handling routines point to two jump addresses located in RAM, the RAM jump vectors, IRQ at $0314 and $0315 and NMI at $0318 and $0319. These RAM vectors contain the actual start address of the interrupt handling routine. The reason for using the RAM vectors is that it allows the programmer to change the starting address of the interrupt handling routines, thus creating his own interrupt handling routines.

Since there are as many as seven different conditions in the VIA which can generate an interrupt request to the processor, a single VIA might require as many as seven interrupt service routines. The VIC, with two VIA chips connected to each of the two interrupt request inputs of the 6502, has a potential requirement for up to fourteen interrupt service routines in the system software (in practice only 6 are used). To determine which one of the seven interrupt conditions

179

of the VIA caused the interrupt request, a programming technique known as polling is used. The interrupt handling routines show how this is done by the VIC.

FUNCTION CONTROL

Control of the various functions and operating modes within the 6522 is accomplished primarily through two registers, the Peripheral Control Register (PCR) and the Auxiliary Control Register (ACR). The PCR is used to select the operating modes of the four peripheral I/O control lines. The ACR selects the operating mode of the two timers and the shift register.

Peripheral Control Register (PCR).
The organisation and location of the Peripheral Control Register is as follows:

Register 13 — Peripheral Control Register
                    VIA No. 1 — Hex $9110   decimal 37148
                    VIA No. 2 — Hex $9120   decimal 37164

Bit No.	7	6	5	4	3	2	1	0
Function	CB2	Control		CB1 Control	CA2	Control		CA1 Control

The PCR has four seperate function fields, each associated with one of the four I/O port peripheral control or 'handshake' lines.

CA1 Control — Bit 0 of the PCR selects the active transition of an input on the CA1 line. If bit 0 of the PCR is set to logic '0', then the CA1 interrupt flag will be set by a negative transition on the CA1 line (the line goes from a logic high to a logic low voltage level). If PCR0 is set to logic '1' then the CA1 interrupt flag is set by a positive transition (low to high).

CA2 Control — Bits 1, 2 and 3 of the PCR. The CA2 line can act as either an interrupt input or a peripheral control output, there are altogether eight different operating modes for this line, they are summarised in the following table:

PCR3	PCR2	PCR1	Mode
0	0	0	Input Mode. set CA2 interrrupt flag on a negative transition of the input signal. Clear IFR0 on a read or write of the Peripheral A Output Register.

0	0	1	Independent interrupt input mode. set IFR0 on a negative transition of CA2 input signal.
			Reading or writing ORA does not clear the CA2 interrupt flag, can only be cleared by writing '1' to the appropriate IFR bit.
0	1	0	Input Mode. Set CA2 interrupt flag on a positive transition of the CA2 input line. Clear IFR0 with a read or write of the Peripheral A Output Register.
0	1	1	Independent interrupt input mode. Set IFR0 on a positive transition of CA2 input signal. Reading or writing ORA does not clear the CA2 interrupt flag, can only be cleared by writing '1' to the appropriate IFR bit.
1	0	0	Handshake output mode. Set CA2 output low on a read or write of the Peripheral A Output Register. Reset CA2 high with an active transition on CA1.
1	0	1	Pulse output mode. CA2 goes low for one processor clock cycle following a read or write of the Peripheral A Output Register.
1	1	0	Manual output mode. The CA2 output is held 'low' in this mode.
1	1	1	Manual output mode. The CA2 output is held 'high' in this mode.

CB1 Control — Bit 4 of the PCR controls the active transition of the CB1 input line in the same manner as that described for the CA1 line. In addition if the Shift Register has been enabled line CB1 will act as an output for the shift register clock pulses. In this mode the CB1 interrupt flag will still respond to the selected transition of the signal on the CB1 line.

CB2 Control — When the serial I/O capability of the shift register is disabled then the function of the CB2 line is controlled by bits 5, 6 and 7 of the PCR. There are altogether eight different operating modes for this line and they are summarised in the following table:

PCR	PCR6	PCR5	Mode
0	0	0	Interrupt input mode. Set CB2 interrupt flag (IFR3) on a negative transition of the CB2 input line. Clear IFR3 on a read or write of Peripheral B Output Register.
0	0	1	Independent interrupt input mode. Set IFR3 on a negative transition of the CB2 input line. Reading or writing ORB does not clear the CB2 interrupt flag, clear by setting IFR3 to '1'.
0	1	0	Input mode. Set CB2 interrupt flag on a positive transition of the CB2 input line Clear the CB2 interrupt flag on a read or write of ORB.
0	1	1	Independent input mode. Set IFR3 on a positive transition of the CB2 input line Reading or writing ORB does not clear CB2 interrupt flag, clear by setting IFR3 to logic '1'.
1	0	0	Handshake output mode. Set CB2 low on a write ORB operation. Reset CB2 high with an active transition of the CB1 input.
1	0	1	Pulse output mode. set CB2 low for one processor clock cycle following a write ORB operation.
1	1	0	Manual output mode. The CB2 output is held 'low' in this mode.
1	1	1	Manual output mode. The CB2 output is held 'high' in this mode.

183

Auxiliary Control Register (ACR).

The organisation and location of the Auxiliary Control Register is as follows:

Register 12 — Auxiliary Control Register
VIA No. 1 — Hex $911B   decimal 37147
VIA No. 2 — Hex $912B   decimal 37163

Bit No.	7	6	5	4	3	2	1	0
Function	Timer 1 Control		Timer 2 Control	Shift Register Control			PB Latch Enable	PA Latch Enable

The ACR controls the operation of six of the 6522 registers, the way in which it controls them is explained in detail in the sections covering those registers.

5 VIC I/O FUNCTIONS

187
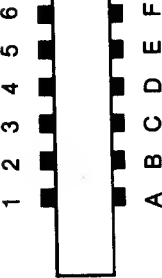
PIN #	TYPE
A-1	GND
B-2	+ 5V
C-3	CASSETTE MOTOR
D-4	CASSETTE READ
E-5	CASSETTE WRITE
F-6	CASSETTE SWITCH

Fig. 25 — The allocation and function of pins on the Cassette connector.
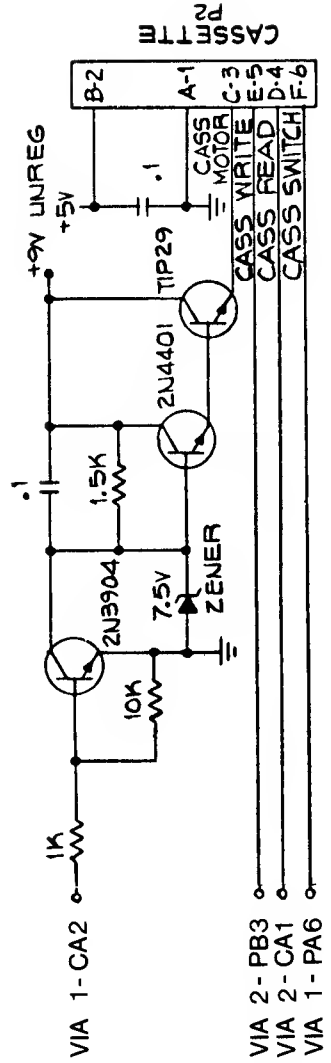
Fig. 26 — The cassette circuit and its connection to the 6522 chips.

THE CASSETTE UNIT.

The Cassette Hardware.
The VIC has a single external cassette unit which is used for program and data storage. The cassette deck is connected to the VIC by six lines — Write, Read, Motor, Sense, and two power lines, around and +5 volts. The connections are shown in Figure. 25. The cassette is controlled by I/O lines from the two VIA chips and the source of each of the cassette control lines from the VIAs is shown in Figure 26. The cassette motor power supply lines are connected to the interface chips via a three transistor driver used to boost the power and voltage, allowing the motor to be driven directly. The output to the motor is an unregulated +9 volts at a power rating of up to 500ma. The cassette deck motor can be turned on and off by toggling the CA2 line on 6522 No. 1.:

POKE 37148, PEEK(37148) AND 241 OR 14   turns the motor on
POKE 37148, PEEK(37148) OR 12 AND NOT 2   turns it off

The sense line input, line PA6 on VIA No. 1., is connected to a switch on the cassette deck which senses when either the Play, Rewind or Fast Forward buttons have been pressed. The switch is only required to sense the pushing of the Play button during a read or write to tape routine, this is done by a subroutine at $F8AB. If either the rewind or fast forward button is pressed accidently instead of the play button the system will be unable to tell the difference and will act as if the play button was pressed. For a similar reason during a record routine the record button must be pressed before the play button since recording will start as soon as the sense switch is closed by pressing the play button.

The cassette "Read" line is connected to the CA1 line of VIA No. 2., and the cassette "Write" line to line PB3 of VIA No. 2. During a Read operation the operating system uses the setting of the CA1 interrupt flag to detect transitions on the cassette Read line. The functioning of the Read and Write lines is controlled entirely by the operating system, the only hardware required being signal amplification and pulse shaping circuitry. These circuits are contained on a small PC board within the cassette deck, their function being to give correct voltage and current to the record head and amplify the input from the read head to give a 5 volt square wave output able to produce an interrupt on the CA1 or CB1 lines.

The Cassette Operation.

In normal usage the cassette deck is assigned an I/O device number, the cassette is device number 1, the device number of the device currently being used is stored in location 186. The device number, the logical file number and the secondary address are used when saving or retrieving data files from the cassette deck. The logical file number can be any number from 1 to 255 and is used to allow multiple files to be kept on the same device, it is of little use with cassette tape and primarily intended for use with floppy disk units. It is usual to have the logical file number the same as the device number, the logical file number of the current file is stored in location 184. The secondary address is important since it determines the operational mode of the cassette, the current secondary address is stored in location 185 the normal default value being zero. If the secondary address is zero then the tape is Opened for a "read" operation, if set to 1 then it is opened for a "write" operation and if 2 then it is opened for a "write" with an end of tape header being forced when the file is closed.

The VIC operating system is configured to allow two different types of file to be stored on cassette: program files and data files. These names are however rather misleading since a program can be stored as a data file and data can be stored as a program file. The difference between these two file types is not in their application but in the way the contents of the machine's memory is recorded. Instead of program and data files we must look upon them as Binary and ASCII files.

A binary file is usually used to store programs, since a binary file is created by the operating system to store the contents of memory between a starting location and an end location. Called a binary file because it stores on tape the binary value in each memory location within the assigned memory area. Basic statements are stored in memory using tokens. The use of tokens means that Basic commands are not stored in the same manner as they are listed on the display or were entered on the keyboard. They are instead stored in memory in a partly encoded form. Being partly encoded, a binary file is a quicker and more efficient way of storing programs. Binary files are essential when saving and loading machine code programs.

The starting address from which a binary file will be saved is stored in locations 172 and 173. These locations are loaded by the Save routine with memory location at which the 'save' will begin, normally they will be set to 0 and 4 thereby pointing to the start of the Basic text

189
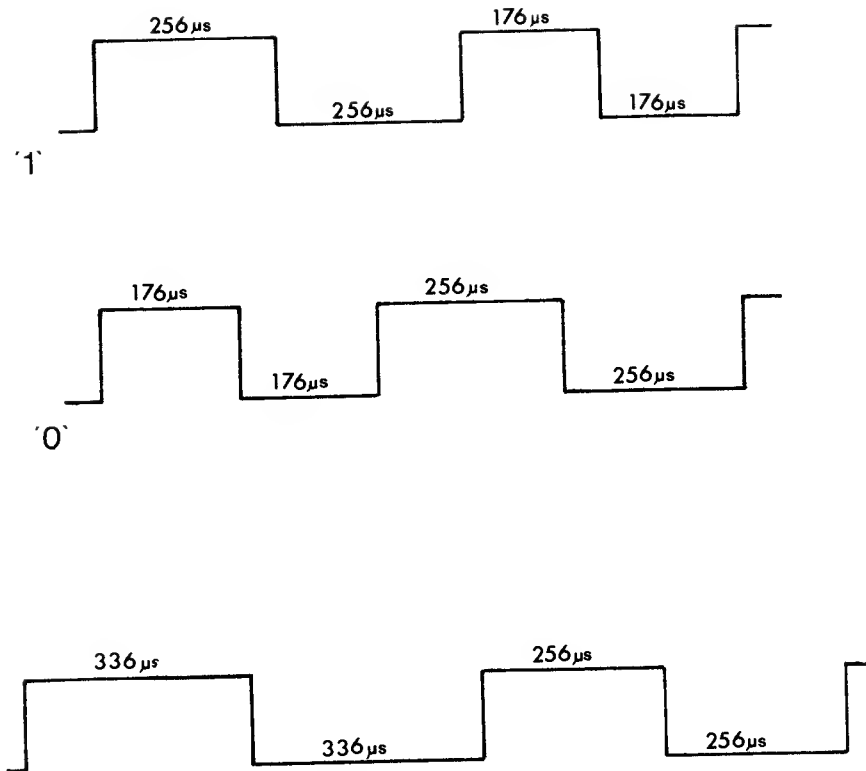
Fig. 27 — The output waveforms to the cassette recorder.

area at 1024. They can be altered by the 'save' routine to point to any location in memory. The end address of the area of memory to be saved is stored in locations 174 and 175. Normally when saving a Basic program these are set to the address of the double zero byte terminating link address. The end address can be altered to any desired location. To change either of these addresses one can not use the normal save routine since this automatically initialises these locations. Instead one must write a small machine code initialisation routine incorporating the desired operating system subroutines. By default a Save command will write a binary file   and a Load command will read a binary file.

ASCII files are normally used to store data, (but can be used to store progams, see Merge procedure) and the format is the same as that displayed on the screen or entered on the keyboard. ASCII files are created or read almost exclusively by instructions from within a Basic program. A binary file is created or read mostly by direct instructions, although the LOAD and SAVE instructions can be used within a program.

An ASCII file must first be opened with an OPEN statement, this specifies the logical file, device number, secondary address and file name. The operating system interprets these parameters and allows the user to read or write the file to the specified device. Data is written to an ASCII file on a particular device with a command to PRINT to the specified logical file number, and data is read by a READ from logical file command.

Whereas a binary file is loaded with the contents of successive memory locations, an ASCII file is loaded with a string of variables. Storing these would require the tape to be turned on and off repeatedly, storing a few bytes of data at a time. The VIC overcomes this by having a 192 byte tape buffer into which all data to be written "to". or read "from" tape is loaded. Only when this buffer is full is the tape motor turned on. Data is stored on tape in blocks of 192 bytes and since the motor is turned on and off between blocks a two second interval is left between blocks to allow the motor to accelerate and decelerate. The beginning of the 192 character buffer starts at address 828. The pointer to the start of the buffer is located at address 178 and 179. The number of characters in a buffer is stored in locations 166. These locations can be used by the programmer to control the amount of space left in a data file. If having opened a file on cassette, the command POKE 166,191 is executed then the contents of the tape buffer even if empty are loaded onto the tape. If records are kept in

191

multiples of 191 bytes we can very easily keep nul or partially filled records allowing future data expansion.

Whether the file being stored is binary or ASCII the recording method is the same involving an encoding method unique to Commodore and designed to ensure maximum reliability of recording and playback. Each byte of data or program is encoded by the operating system using pulses of three distinct audio frequencies, these are long pulses with a frequency of 1488Hz, medium pulses at 1953Hz and short pulses at 2840Hz. All these pulses are square waves with a mark space ratio of 1:1, one cycle of a medium frequency is 256 microseconds in the high state and 256 microseconds in the low state. The operating system takes about 9 milliseconds to record a byte of data consisting of the eight data bits, a word marker bit and an odd parity bit. The data bits are either ones or zeros and are encoded by a sequence of medium and short pulses: a "1" is one cycle of a medium length pulse followed by one cycle of a short length pulse and "0" is one cycle of a short length pulse followed by one cycle of a medium length pulse. Each bit consists of two square wave pulse cycles, one short and one medium with a total duration of 864 microseconds. The waveform timing is shown in the diagram in Figure 27.

The 'odd parity' bit is required for error checking and is encoded like the eight data bits using a long and short pulse, its state is determined by the contents of the eight data bits. The 'word marker' separates each byte of data and also signals to the operating system the beginning of each byte. The word marker is encoded as one cycle of a long pulse followed by one cycle of a medium pulse, see Figure 27.

Since a byte of data is recorded in just 8.96 milliseconds, a 192 byte block of data in an ASCII file should be recorded in just over 1.7 seconds. However, on timing such a recording we find it takes 5.7 seconds. There are two causes for this discrepency in timing. Firstly to reduce the possibility of audio dropouts the data is recorded twice. Secondly a two second inter-record gap is left between each record of 192 bytes.

The extensive use of error checking techniques is one reason why the tape system on the VIC is so much better than that available on most other popular computers. There are two levels of error checking. The first divides the data into blocks of eight bytes and then computes a ninth byte, the checksum digit. The checksum is obtained by adding the eighth data bytes together, the checksum is the least significant byte of the result. On reading the tape if one bit in the eight bytes is

dropped and a zero becomes a one or vice versa the checksum can be used to detect this error. To do this the same procedure to calculate the check digit is performed, the result will be different to that stored in byte nine, the check digit of that block computed when the tape was recorded. The second level of error checking involves recording each block of data twice. This allows errors detected by the check digit to be corrected during the second reading of the 192 byte data block. By recording the data twice a verification can be performed by comparing the contents of the two blocks, this will detect the few errors not detected by the checksum.

The use of pulse sequence rather than two frequencies as in a standard FSK recording has a great advantage since it allows the operating system to easily compensate for variations in recording speed. Normally a hardware phase locked loop circuit would be used to lock the system onto the correct frequencies coming from the tape head, the VIC however uses software to perform this process. A ten second leader is written on the tape before recording of the data or program commences. This leader has two functions, first it allows the tape motor to reach the correct speed and secondly the sequence of short pulses written on the leader is used to synchronise the read routine timing to the timing on the tape. The operating system can thus produce a correction factor which allows a very wide variation in tape speed without affecting reading. The system timing used to perform both reading and writing is very accurate, based as it is on the crystal controlled system clock and Timer 1 and Timer 2 of VIA No. 2. Inter-record gaps are only used in ASCII files and their function is to allow the tape motor time to decelerate after being turned off and accelerate to the correct speed when turned on prior to a block read or write. Each inter-record gap is approximately two seconds long and is recorded as a sequence of short pulses in the same manner as the ten second leader. There is also a gap between blocks, when the first block of 192 bytes is recorded it is followed by a block end marker, which consists of one single long pulse followed by 50+ cycles of short pulses then the second recording of the 192 block starts, this is identical to the first block.

The first record written on the tape after the ten second leader in both ASCII and binary files is a 192 character file header block. The file header contains the name of the file, the starting memory location, and the end location. In an ASCII file these addresses are the beginning and end of the tape buffer, in a binary file they point to the area of memory in which the program is to be stored.

193

The file name can be up to 128 bytes long, the length of the file name is stored in location 183, and when read is compared with the requested file name in the Load or Open command. If the name is the same then the operating system will read the file, if different then it will search for the next ten second interfile gap and another header block. The file name is stored during a read or write operation in a block of memory, the starting address of which is stored locations 187 and 188. On completion of the operation these are reset to point to a location in the operating system. The starting location is normally set to the beginning of the user memory area, address 1024, however it can be changed to point to any location, a method employed when recording programs in a machine code using the monitor, and also in the no copy program. The starting address is pointed to by the contents of locations 172 and 173, the end address being stored in locations 174 and 175. Normally this is the highest byte of memory occupied by the program, however it can be altered to point to any address providing it is greater than the start address.

Important Memory Locations Used by the VIC Cassette.

$92	— temp used to adjust software servo
$93	— verify or load lag (0 = loading)
$96	— flags if we have block sync (16 zero dipoles)
$9B	— holds currently calculated parity bit
$9C	— cassette dipole switch
$9E	— count of read locations in error pointer into $0100
$9F	— count of re-read locations during pass No. 2.
$A4	— used to indicate which half of dipole we are in
$A5	— countdown for tape write; sync on tape header
$A6	— cassette buffer pointer
$A7	— tape short count
$A8	— flags errors (if zero then no error)
$A9	— counts zeros (if zero then correct No. of dipoles)
$AA	— bits 6 & 7 hold function mode, rest = sync countdown
$AC-$AD	— indirect address to start of tape data storage
$AE-$AF	— indirect address to end of tape data storage
$B1	— holds dipole time during types calculations
$B2-$B3	— start address of tape buffer
$B4	— flags if we have a byte sync (a longlong)
$B5	— used to preserve sync outside of bit routines
$B6	— has combined error values from bit routines
$B7	— length of current file name string
$B8	— current logical file number

```
$B9          — current secondary address
$BA          — current device number
$BB-$BC      — address of current file name string
$BD          — receive input character
$BE          — indicates which block we are looking at (0 to exit)
$BF          — holds input byte being built
$CO          — cassette manual/controlled switch
$C3-$C4      — cassette load temp storage
$D7          — holds most recent dipole bit value
$0100-$01FF  — storage of bad read locations, bottom of stack
$0259-$0262  — logical file number table
$0263-$026C  — device number table
$026D-$0276  — secondary address table
$033C-$03FC  — cassette buffer
```

System subroutines used by the VIC cassette.

$F542-$F646 — Load RAM routine. Loads from cassette or serial device as determined by contents of $BA. Verify flag in .A. Alternately load if $B9 = 0 (normal $B9 = 1) .X, .Y contain load address if .A = 0 performs load (0 is verify).

High load address returned in .X and .Y.

$F675-$F734 — Save RAM routine. Saves to cassette or serial device selected by contents of $BA. Start of save is indirect at .A, end of save is .X, .Y.

$F7AF-$F889 — Find tape header information. reads tape until one of the following block types is found: basic data file header, or basic load file. For success carry is clear on return. In addition accumulator is 0 if stop key was pressed.

$F88A-$F98E — Miscellaneous tape control routines.
Includes:
```
  F8AB    — cassette sense switch control
  $F8B7 — check for play and record
  $F8C0 — read header block entry
  $F8C9 — read load block entry
  $F8E3 — write header block entry
  $F8F4 — start tape operation entry point
  $F95D — set up timeout watch for next dipole
```

$F98E-$FABC — Cassette read routines. The character read is passed to the byte routine in location $BF.

$FABD-$FBE9 — Byte handler for cassette read. The byte assembled from reading tape is passed to this routine in $BD. $A8 is set if byte read is in error and $A9 is set if the interrupt program is reading zeros. $AA tells us what we are doing, bit 7 says ignore bytes until $A9 is set and bit 6 says load the byte. Otherwise $AA is a countdown after sync. If $93 is set we do a compare instead of store and set status. $BE counts the two blocks, $9E is the index to the error table for pass No. 1., and $9F is index to correction table for pass No. 2.

$FBEA-$FD21 — Cassette write routines. Location $BE is the block counter for record. If $BE = 2 then first header
  = 1 first data
  = 0 second data

Note: The IRQ vectors are changed during cassette operation, if the user has reset these vectors then they should be restored to their normal value prior to using the cassette.

THE VIC KEYBOARD

The VIC keyboard has a total of 66 keys, this comprises 64 alphanumeric and special function keys, the restore key and a shift lock key. The 64 alphanumeric and special function keys are connected as a simple matrix to the two eight line I/O ports of VIA No. 2., the way they are connected is shown in Figure 29. The restore key connects the CA1 line of VIA No.1., to ground and is used to generate an NMI processor interrupt. The shift lock is simply a mechanical device for keeping the shift key depressed. The keyboard matrix is scanned for a key depression by having one eight line I/O port configured as outputs and the other as inputs. Each output line is connected via key switches to all eight of the input lines. If one of those eight keys is depressed then the voltage level on the output line will be transferred to the input line corresponding to that key. By having a scanning sequence where each of the eight output lines in turn go 'low' while the rest stay 'high' the operating system software can determine which key in the 64 key matrix is currently pressed.

The scanning of the keyboard matrix and testing for depression of the restore key are all under software control. The entire processor time can not be devoted to keyboard scanning therefore scanning is initiated by a regular 1/60 second interrupt. Keyboard scanning is one of the functions of the IRQ interrupt servicing routine. The 1/60 second regular interrupt is generated by Timer 1 of VIA No. 2. The interrupt service routine starts at location $EABF and the keyboard scanning portion at $EB1E.

The keyboard scanning routine goes through a sequence of operations the result of which is to place each input character into a special section of memory, the keyboard buffer. The sequence is as follows:

1 — check if key pressed, if not then exit from routine.

2 — initialise I/O ports of VIA No. 2., for keyboard scan and set pointers into keyboard character table No. 1., set character counter to 0.

3 — set one line of port B low and test for character input on port A by performing eight right shifts of the contents of port A register, if carry is set then character present. Each shift increments character count, store character counter in .Y.

197
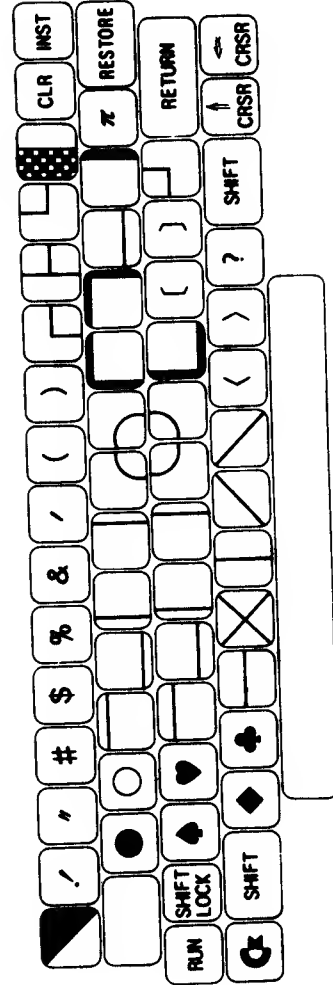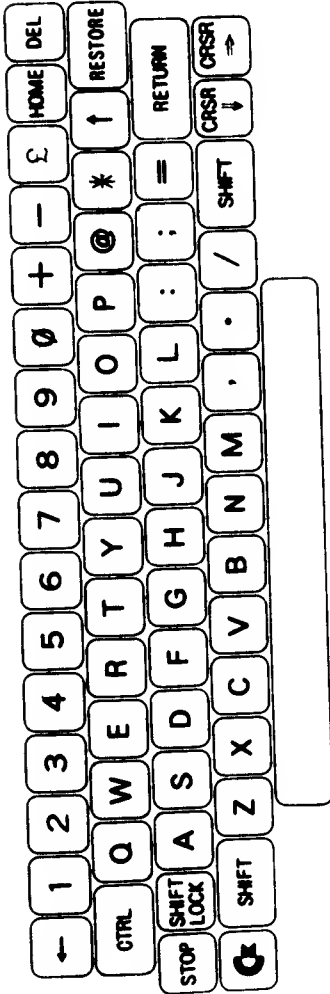
Fig. 28 — Layout of the VIC keyboard (shifted and unshifted).

4 — go back to step 3 and repeat for next column, if character found then continue.

5 — use character count value as index pointer into keyboard character table to get ASCII code corresponding to depressed key.

6 — see if it is a Shift or Stop key.

7 — evaluate Shift function.
  if Shift key pressed then use character count to access keyboard character table No. 2.
  if CBM key pressed then use character count to access keyboard character table No. 3.
  if CBM and Shift pressed then use character count to access keyboard character table No. 4.

8 — use character count value as index pointer into keyboard character table designated in step 7.

9 — check for repeat key operation.

10 — check for screen editor keys and take appropriate action.

11 — do repeat if required.

12 — put ASCII character obtained from the keyboard character tables into the keyboard buffer, increment the pointer into the keyboard buffer.

The contents of the 10 character keyboard buffer are accessed on a first in first out basis by the screen handling routines. These routines take the first character in the keyboard buffer, decrement the buffer pointer and close up the buffer by moving the contents down one byte thereby leaving space for new input characters. The exact function of the screen handling routines depends on the mode of operation of the VIC. If the VIC is in the Direct mode then the keyboard input is part of Basic routine to receive a program line from the keyboard, the starting address of this routine is $C560. If the VIC is running a Basic program then keyboard input is part of the Basic character string input routine, starting at location $CBBF.

In the Direct mode, characters are removed from the keyboard

199
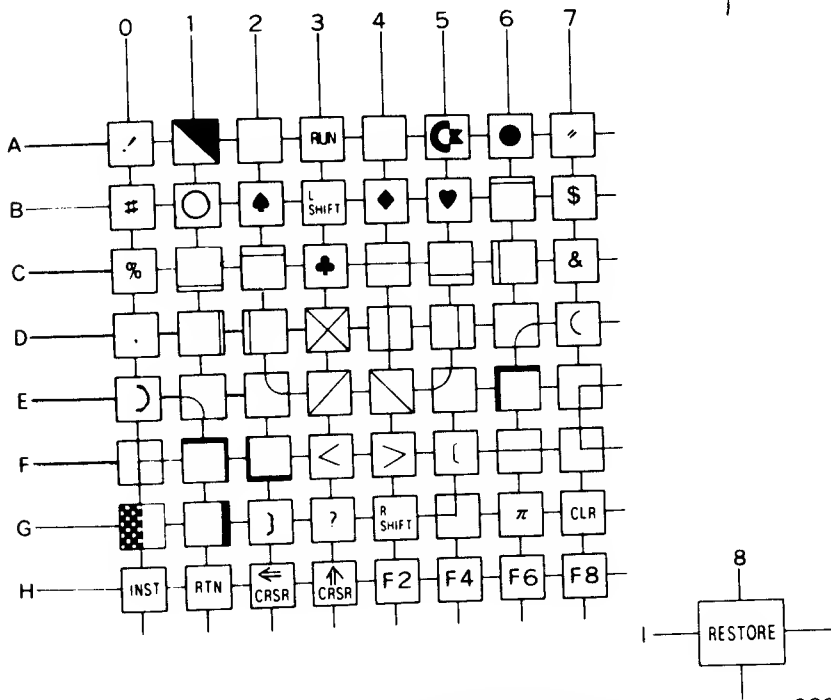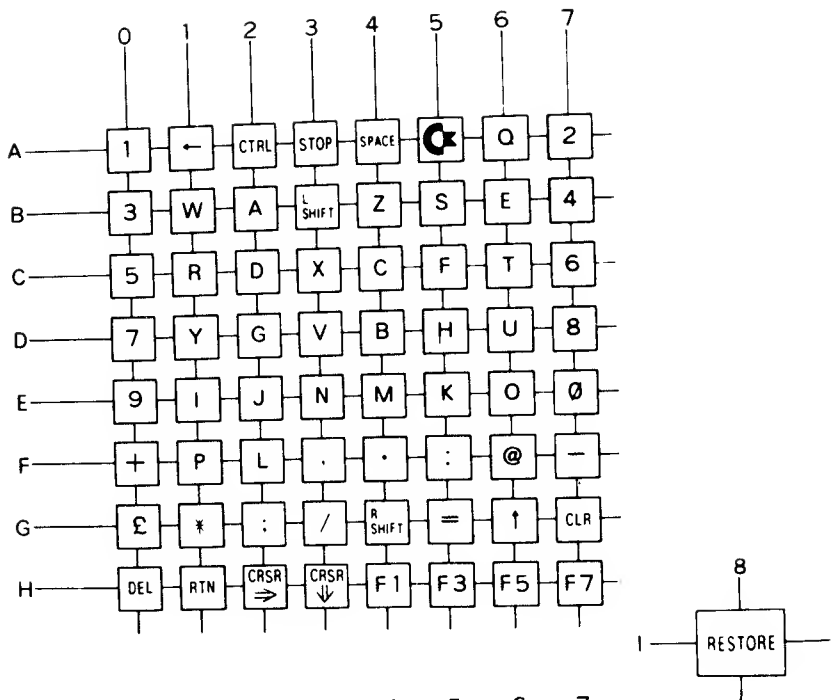
Fig. 29 — Matrix of keyboard connections to 6522 I/O lines.

200

buffer and displayed or the screen, the character is also placed in the 88 byte Basic input buffer. This continues until a carriage return key is pressed. The Basic interpreter then checks the contents of the Basic input buffer to determine if the input is a valid command, and if so then executes the command, if not returns a syntax error message.

In the program mode, characters are removed from the keyboard queue when required by the Basic INPUT or GET commands during the running of a program. It is in this mode that the keyboard buffer is most useful, it allows a flexibility in timing between input from the scanning routine, and the execution by the Basic interpreter of the INPUT or GET command. This does have its drawbacks since any characters in the keyboard buffer prior to the execution of the input will be accepted as valid input characters. This can give rise to spurious data input and can be avoided by clearing the keyboard queue pointer (location 649 is set to zero).

One of the disadvantages of using the interrupt to initiate keyboard scanning is that the interrupt routines are changed during I/O operations between the processor and the cassette or serial I/O devices. The drawback is that the user looses control of the system if the keyboard is disabled as a result of a temporary sessation of scanning. The principle control key required is the Stop key, this allows the user to exit from an I/O 'hangup'. The solution to this problem is to leave the keyboard after each scan so that the column containing the Stop key is still being scanned, this means that output line PB3 is left in an 'on' state. The I/O routines can then very simply test for a depression of the Stop key by reading the input register of port A on VIA No. 2. The Stop key routine is thus separated from the rest of the keyboard scanning routines, the Stop key routine is in two sections:

$F755 — this is part of the time function routine which is called in all IRQ servicing routines, it updates the real time clock and checks the Stop key. The contents of Port B of VIA No.2., are read, debounced to make sure that the contents are stable and then stored in the Stop key flag — location $91.

$F770 — this part of the routine is the main Stop key routine, it is called by an indirect address stored in the Stop key RAM vector at location $0328. This routine takes the contents of location $91 and compares it with the value $FE if equal then the Stop key has been

```
1 REM *ROUTINE TO TEST WHICH FUNCTION
2 REM *KEY HAS BEEN PRESSED
3 REM
4 REM
8 REM *WHICH KEY PRESSED?
9 REM
10 A=PEEK(203)
11 REM
12 REM *SHIFY KEY DOWN?
14 REM
15 B=PEEK(653)
16 REM
17 REM *DECODE KEY NUMBER
18 REM
20 K=0
25 IFA=39THENK=1:GOTO50
30 IFA=47THENK=3:GOTO50
35 IFA=55THENK=5:GOTO50
40 IFA=63THENK=7:GOTO50
45 GOTO10
50 IFB>1THENB=0
55 K=K+B
60 PRINT"FUNCTION KEY"K"PRESSED"
65 GOTO10
```

Fig. 30 — Program to use VIC function keys.

depressed. This initiates the clearing of the keyboard queue and I/O channels prior to returning to direct mode operation. If the Stop key has not been pressed then control returns to the calling routine.

$FEA9 — routine to handle an NMI interrupt, this is generated if the "Restore" key is pressed, the "restore" function is only initiated if the "Stop" key is also pressed. If both keys are pressed then a Basic warm start is initiated by a jump to $0002.

Important memory locations used by the VIC keyboard.

$91	— Stop key flag set if Stop key depressed
$C5	— Key scan index
$C6	— Index to keyboard queue
$F5-$F6	— Indirect jump address to keys on table
$0200-$0258	— Basic input buffer
$0277-$0280	— Keyboard buffer
$028A	— Key repeat flag
$028D	— Shift flag
028F-$0290	— Indirect jump address for keyboard table
$0291	— VIC mode (CBM key pressed?)

System subroutines used by keyboard.

$C560 — routine to get Basic command from the keyboard and place in the Basic input buffer ready for the interpreter.

$CBBF — routine performs the Basic character string input function.

$E5CF — remove character from keyboard queue and return in .A

$E64F — input line until carriage return key is pressed, part of Basic input routine.

$E742 — displays character in .A on the screen at the current cursor location.

$E800 — handles shift keys

$EABF — IRQ service routine

$EB1E — general keyboard scan, uses keyboard character tables to obtain correct ASCII code for character and puts character into keyboard queue.

$EBDC — shift key logic

$EC5E — start of keyboard character table No. 1.

$EC9F — start of keyboard character table No. 2.

$ECE0 — start of keyboard character table No. 3.

$ED72 — start of keyboard character table No. 4.

203

RS232 SERIAL COMMUNICATIONS

The VIC is able to communicate with peripheral devices; printers, modems etc, using a serial communications port, known as an RS232 I/O port. The name RS232 simply refers to an industry standard form of serial communication for computing devices. A serial I/O port can consist of as few as three lines, an output or transmit line, an input or receive line and a common ground line. The data is transmitted or received as a stream of pulses, a single byte becomes a string of eight pulses.

Although a serial port can have just three lines, other lines are frequently used to transfer control information. The VIC is able to receive and generate such control signals to implement a full 'X line' interface as well as the simple '3 line' interface. Whichever implementation is used all the lines are connected to I/O Port B of VIA No. 1., the same lines used for the user port. Normally an RS232 interface card will be used to connect between the parallel port and a standard RS232 connector, the card will also provide buffering and a higher drive voltage. For communications using the simple 3 line mode an interface card can easily be constructed using a couple of buffer/driver ICs. The RS232 line normally transmits data using a 12 volt signal, however, providing cables are kept short it will work with a 5 volt signal. The standard RS232 connector is shown in Figure 32, the function and pin assigment of each of these lines is as follows:

VIA line No.	RS232 pin No.	VIA pin No.	Abv	EIA	In/Out	Modes	Function
GND	1	A	GND	AA	—	1,2	Protective ground
CB1	3	B	SIN	BB	In	1,2	Received data
PB0	3	C	SIN	BB	In	1,2	Connected to SIN
PB1	4	D	RTS	CA	Out	2	Request to send
PB2	20	E	DTR	CD	Out	2	Data terminal ready
PB3	18	F	RI	CE	In	3	Ring indicator
PB4	8	H	DCD	CF	In	2	Received line signal
PB6	5	K	CTS	CB	In	2	Clear to send
PB7	6	L	DSR	CC	In	2	Data set ready
CB2	2	M	SOUT	BA	Out	1,2	Transmitted data
GND	7	N	GND	AB	—	2,3	Signal ground

Modes:
1 — 3 -line interface (note RTS and DTR are both held high during this mode).
2 — X-line interface.
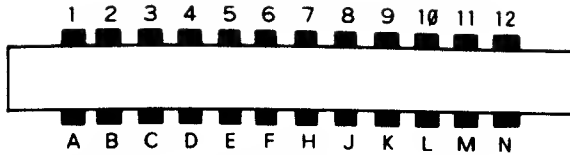
PIN	TYPE	NOTE	PIN	TYPE	RS232 FUNCTION
1	GND		A	GND	
2	+5V	100mA MAX	B	CB1	
3	RESET		C	PB0	⊥ SIN
4	JOY0		D	PB1	— RTS
5	JOY1		E	PB2	— DTR
6	JOY2		F	PB3	— RI
7	LIGHT PEN		H	PB4	— DCD
8	CASSETTE SWITCH		J	PB5	
9	SERIAL ATN IN		K	PB6	— CTS
10	+9V	100mA MAX	L	PB7	— DSR
11	GND		M	CB2	— SOUT
12	GND		N	GND	— GND

Fig. 31 — VIC RS-232 connector and pin allocations.

PIN		
1	Protective Ground	AA
2	Transmitted Data	BA
3	Received Data	BB
4	Request To Send	CA
5	Clear To Send	CB
6	Data Set Ready	CC
7	Signal Ground	AB
8	Carrier Detect	CF
9	(not used)	
10	"	
11	"	
12	"	
13	"	
14	"	
15	"	
16	"	
17	"	
18	"	
19	"	
20	Data Terminal Raady	CD
21	(not used)	
22	"	
23	"	
24	"	
25	"	

Fig. 32 — Standard RS-232 connector and EIA line coding.

205

3 — User available only and not implemented or used in the VIC RS232 code.

The implementation of the RS232 port on the VIC is very interesting since it involves the use of software to emulate a hardware device. The hardware is the 6551 Universal Asynchronous Transmitter and Receiver or UART. It was originally intended by the VIC designers to use this chip to generate the RS232 I/O, however, MOS were unable to deliver usable devices in time for the VIC production, and software emulation had to be employed. An exact emulation of the function of the 6551 is used since this allows the manufacturers to change the VIC hardware design to incorporate the 6551 as soon as this device becomes available. Like the other I/O chips the 6551 functions are controlled by registers at specific memory locations. The pseudo 6551 registers are located in various parts of the variable storage area at the bottom of VIC memory. Besides the registers, the RS232 operating routines require two 256 byte buffers, one for received data and the other for transmitted data. The 512 bytes of memory occupied by these buffers is located at the top of available RAM memory, the starting address of the two buffers is stored in four register bytes. The two most important registers are the Control, and Command Registers, these determine the exact operation of the RS232 port, they can be summarised as follows:

The 6551 Pseudo Control Register — Hex $0293 decimal 659

The function of the Control register is to set the speed of data transmission and reception and set the number of bits needed to transmit each character. The speed at which data is input or output is called the baud rate, and the value assigned to this is the number of bits per second. If the baud rate is set to 300 baud, and each character is transmitted as the eight character bits plus one stop bit and one parity bit — total of ten bits — then 30 characters will be transmitted every second. The selected baud rate depends on the specifications of the device communicating with the VIC via the RS232 port, check the manual of the device before setting this value. Bits 5, 6 and 7 control the number of bits needed to transmit or receive data between the VIC and a peripheral. The number of bits per character plus the number of stop bits depends on the device communicating with the VIC via the RS232 port.

The 6551 Pseudo Command Register — Hex $0294 decimal 660

The Command Register controls the mode of data transmission and

7		6	5		4		3	2	1	0	

STOP BITS
0 = 1 STOP BIT
1 = 2 STOP BITS

WORD LENGTH

BAUD RATE

BIT		DATA
6	5	WORD LENGTH
0	0	8 BITS
0	1	7 BITS
1	0	6 BITS
1	1	5 BITS

0	0	0	0	USER RATE (NI)
0	0	0	1	50 BAUD
0	0	1	0	75
0	0	1	1	110
0	1	0	0	134.5
0	1	0	1	150
0	1	1	0	300
0	1	1	1	600
1	0	0	0	1200
1	0	0	1	1800 2400
1	0	1	0	2400
1	0	1	1	3600 (NI)
1	1	0	0	4800 (NI)
1	1	0	1	7200 (NI)
1	1	1	0	9600 (NI)
1	1	1	1	19200 (NI)

UNUSED BIT

(NI) Not implemented in the VIC-20 system
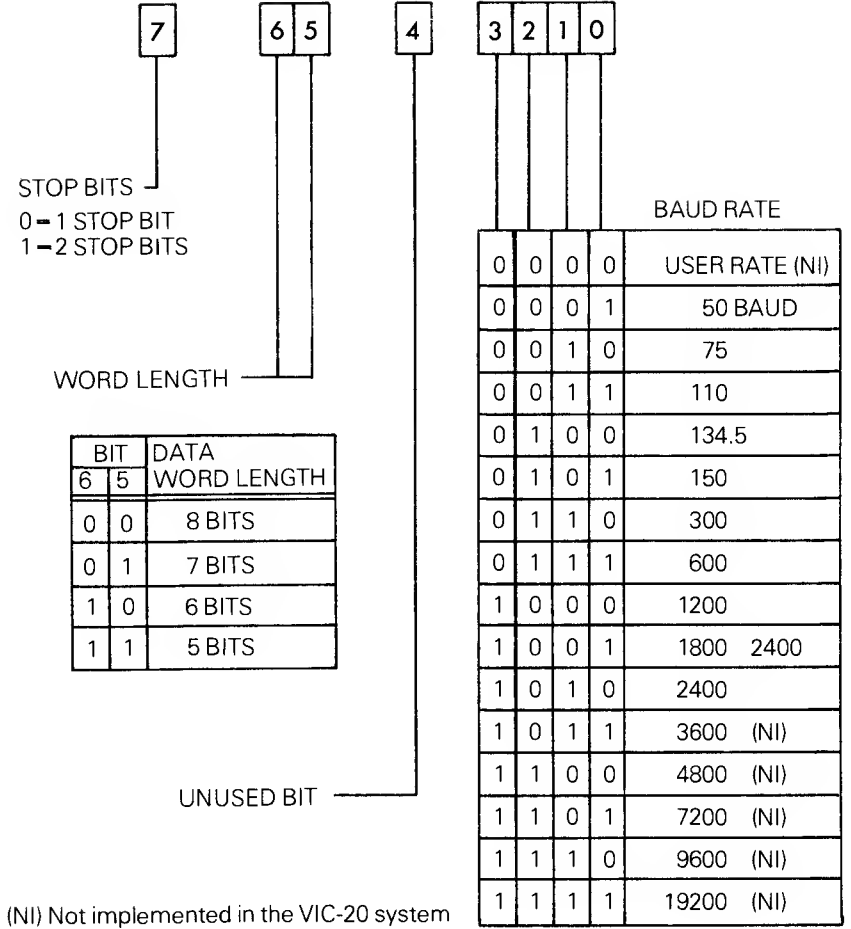
Fig. 33 — Function of bits in VIC RS-232 Control register.

207

reception. Bit 0 sets the mode, a 3 line mode or a X line mode. Bit 4 sets the Duplex mode as follows:

Full Duplex — simultaneous transmission and reception of data.
Half Duplex — alternate transmission and reception of data.

Bits 5, 6 and 7 determine the nature of the parity bit and whether the mark or space is transmitted. The parity bit is transmitted after the data bits and has an error checking function, the choice of whether the parity is disabled or is set to odd or even depends on the specification of the communicating device attached to the VIC RS232 port. The mark/space setting determines whether a logic '1' is transmitted as a zero voltage or a positive voltage, this is shown in Figure 4.

The RS232 Status register — Hex $0297 decimal 663

The other memory locations and pseudo 6551 registers are as follows:

$A7	— receiver input bit temporary storage
$A8	— receiver bit count in
$A9	— receiver flag Start bit check
$AA	— receiver byte buffer/assembly location
$AB	—receiver parity bit storage
$B4	— transmitter bit count out
$B5	— transmitter next bit to be sent
$B6	— transmitter byte buffer/disassembly location
F7-$F8	— a two byte pointer to the receiver buffer base location
$F9-$FA	— a two byte pointer to the transmitter buffer base location
$0298	— the number of bits to be sent/received
$0299-$029A	— the time for transmission of one bit cell based on system clock/baud rate
$029B	— the byte index to the end of the receiver FIFO buffer
$029C	— the byte index to the start of the receiver FIFO buffer
$029D	— the byte index to the start of the transmitter FIFO buffer
$029E	— the byte index to the end of the transmitter FIFO buffer
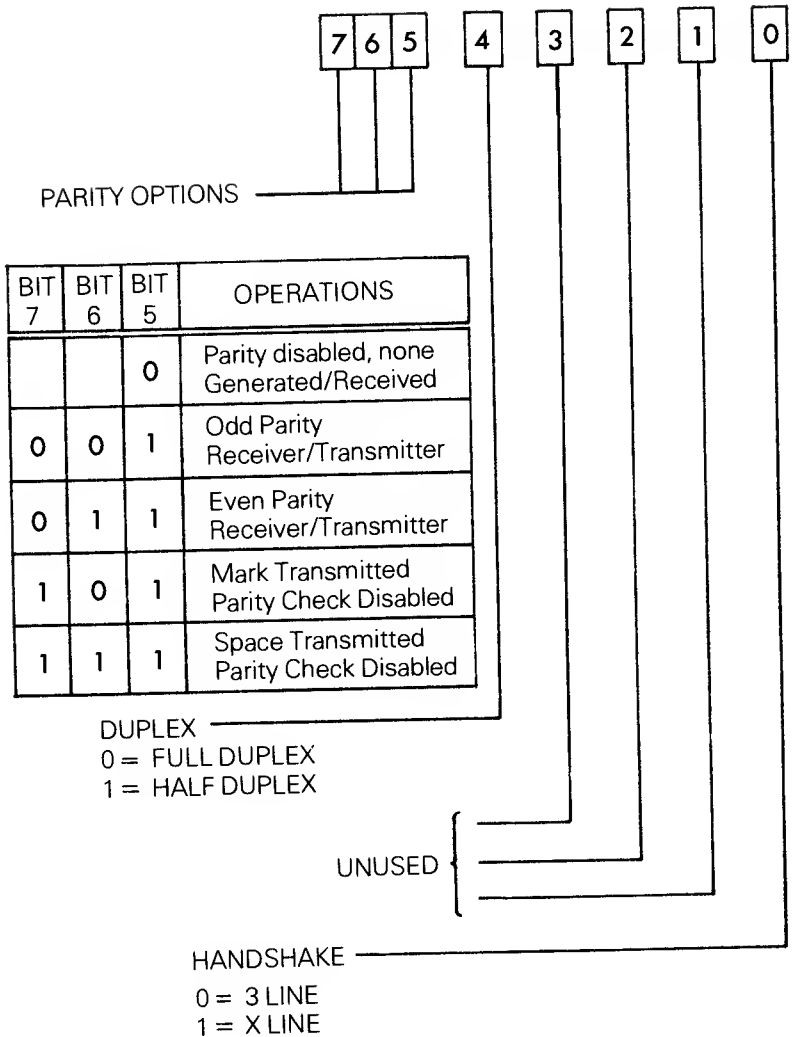
Fig. 34 — Function of bits in VIC RS-232 Command register.

RS232 System Routine Entry Points

$EFA3 — entry for NMI continue routine
$EFBF — calculate parity, $B5 = 0 upon entry
$EFE8 — count stop bits
$EFEE — entry to start of byte transmission
$EFFB — set up to send next byte
$F016 — set errors
$F027 — calculate No. of bits to be sent, returns No. bits + 1
$F036 — NMI routine to collect data into bytes
$F040 — calculate parity
$F046 — shift data bit in
$F04B — have stop bit so store in buffer
$F05B — enable to receive a byte
$F068 — receiver start bit check
$F06F — put data in buffer (at parity time)
$F08B — parity checking
$F094 — check calculated parity
$F09F — errors reported
$F0BC — output a file over user port using RS232
$F0C4 — check for DSR and RTS
$F0CD — check for active input, RTS will be low if currently
          inputting
$F0D4 — wait for CTS to be off
$F0D9 — turn on RTS
$F0E1 — wait for CTS to go on
$F0ED — buffer handler to output a character
$F0FC — set up if necessary to output
$F102 — set up for a first byte out
$F10E — set up for T1 NMI's
$F116 — input a file over user port using RS232
$F122 — check if DSR and not RTS
$F12B — wait for active output to be done
$F130 — turn off RTS
$F138 — wait for DCD to go high
$F13F — enable CB1 for RS232 input
$F146 — if not 3 line half then see if we need to turn on CB1
$F14F — input a character buffer handler
$F15C — receiver always runs
$F160 — protect serial/cassette from RS232 NMI's
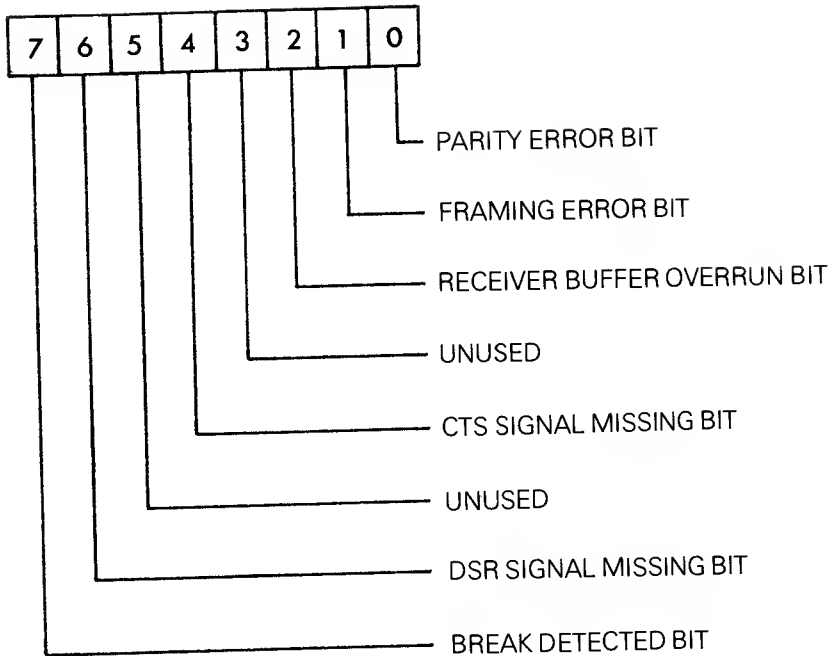
RS-232 STATUS REGISTER — $0297

Fig. 35 — Function of bits in VIC RS-232 Status register.

Using the RS232 Port

Opening an RS232 Channel

Basic Syntax: OPEN   lf, 2, 0, "(control register) (command register)"

lf — Normal logical file ID (1-255). If lf)127 then line feed follows carriage return

(control register) — an ASCII character equivalent to the required bit setting of the Control Register. Example: to set baud rate to 300 and transmit 7 bit code use CHR$(6+32) — this sets bits 1, 2 and 5 to logic '1' and leaves the rest at logic '0'.

(command register) — an ASCII character equivalent to the required bit setting of the Command Register. Example: to set the output to mark parity and full duplex use CHR$(32+128) — this sets bits 5 and 7 to logic '1' and leaves the rest at logic '0'.

Machine Code Entry Point: Hex $FFC0

Notes on Usage: Only one RS232 channel should be open at any time, since the OPEN statement resets the buffer pointers, a second OPEN will destroy any data in the buffers set up in the first OPEN. The OPEN RS232 channel command should be used before any variable or DIM statements, failure to do this will cause wiping of data. This is because the OPEN RS232 channel command performs an automatic CLR before allocating the 512 bytes at the top of memory used for the two RS232 data buffers. If there is insufficient space at the top of memory for the 512 byte buffer then program destruction will result. The file name field in the OPEN command statement can have up to four characters, only two characters are currently used by the system (see Basic syntax) the other two characters are for future systems options. No error checking is done by the system on the contents of the control or command characters, errors in baud rate selection will cause system malfunction. A non-implemented baud rate will cause an index to bad page data, and output will be set to a rate below 50 baud.

Receiving Data From an RS232 Channel.

Basic Syntax: GET # If, (string variable)

If — logical file ID used in OPEN RS232 channel command

Machine Code Entry Points:

$FFC6 — Open channel for Input. Handles full X-line implementation according to EIA standard RS232C interfaces. The RTS, CTS and DCD lines are implemented when the VIC is designated as a Data Terminal device.

$FFE4 — Get character from buffer

Notes on Usage: Received data is put into the VICs 255 byte internal receiver buffer set up during the OPEN RS232 channel command. Data input is under control of the 6522 timers and interrupts and is performed in the background during the running of a Basic program. This is done by having the RS232 data input line connected to the CB1 handshake line, an input on CB1 will generate an NMI system interrupt. The use of NMI interrupts is the reason why the cassette and serial bus should not be used during RS232 data communications. The NMI interrupt will call the serial data input routines whenever data is present on the RS232 input. These routines will place the received data into the 255 byte receiver buffer located at the top of RAM memory. If the input data has a word width less than eight bits then all unused bits will be filled with zero.

The receiver buffer is organised as a First In First Out — FIFO — buffer. The buffer removes the necessity for Basic to wait for data input before processing each byte of data. Instead the Basic program can take data from the buffer when it needs it rather than when it is presented. Basic accesses the buffer using the GET command to transfer a single byte of data into a Basic variable. If there is no data in the buffer then the GET # command will return with a null character. If the buffer should overflow then all characters received during the overflow condition are lost, an overflow condition is indicated by bit 2 in the RS232 Status register being set. An overflow condition will frequently result, if an attempt is made to input data at fairly high data rates using Basic. This is because Basic is normally slow and the use of the GET command with string concattination will give rise to frequent garbage collects. Machine language routines are best used for data rates above the normal 300 baud.

213

Transmitting Data to an RS232 Channel

Basic Syntax: CMD lf
PRINT $\#$ lf, (variable list)

lf — logical file ID set up in the OPEN RS232 channel command

Machine Code Entry Points:

$FFC9 — Open channel for output. This handles X-line handshaking for the implementation of an EIA standard RS232 interface. The RTS, CTS, and DCD lines are implemented with the VIC as a Data Terminal.

$FFD2 — Output character to channel

Notes on Usage: When either one of the two Basic commands are used data is first transferred from the assigned string or memory block to the 255 byte transmitter buffer. From here it is output to the RS232 channel using the format and baud rate assigned in the OPEN RS232 channel command. Data output is transparent to the operation of Basic since the timing is done by the 6522 timers and output of each byte initiated by an NMI system interrupt. As with data input on the RS232 the cassette or serial IEEE port should not be used during data transmission on the RS232 otherwise interrupt conflicts will occur. There is no carriage return delay implemented by the output channel, therefore a normal RS232 printer cannot correctly output the data, unless some form of internal buffering or other hold-off is implemented by the printer. If a CTS handshake is implemented (in the X-line mode) then the VIC buffer will fill, and output will not occur until transmission is allowed by an input on CTS.

Closing an RS232 Data Channel.

Basic Syntax: CLOSE lf

lf — logical file ID set up in the OPEN RS232 channel command

Machine Code Entry Points

$FFC3 — Close logical file

Notes on Usage: Closing the RS232 file causes all the data in the buffers to be discarded, stops data transmitting or receiving, sets the

RTS and SOUT lines high, and de-allocates the memory area used for the RS232 buffers. Closing the RS232 file will also allow the cassette or serial IEEE ports to be used. Before closing the channel care should be taken to ensure that all data in the buffer is transmitted. This can be done by checking the status (ST variable is = 0) and that bit 6 of parallel Port A of VIA No. 1 location 37151 is set to logic 1, if both are true then there is still data in the buffer.

THE JOYSTICKS

Two different types of joysticks can be attached to the VIC, a simple paddle switch joystick, and a potentiometer joystick. The principle application for joysticks is in interactive games and simulation programs. The joystick is used to control the position of an object on the screen, this can be either the cursor or a special graphics character or characters. Alternatively the cursor can be used to change the viewing position, using the joystick like the control stick on an aircraft. The choice of which type of joystick is used depends on whether fine positional control or simple left, right, forward or backward input is required. If fine positional control is required where a particular joystick position has a unique value, then a potentiometer joystick is required. If simply telling the computer the direction of the joystick movement, using one of eight directions is adaquate, then a switch joystick is the best choice.

Switch Joystick

A switch joystick consists of four switches mounted at right angles to each other. The joystick handle is connected to a mechanism which allows no more than two adjacent switches to be closed at any one time. The joystick handle has nine possible positions:

one with no switches closed — the handle is vertical

four positions with one switch closed — handle in north, south, east, and west positions.

four positions with two switches closed — handle in north east, south east, south west, and north west positions

An extra switch is usually mounted on the end of the joystick handle, called the 'Fire button'. This is usually used to indicate to the computer when the cursor or games figure is in the right position on the screen. Each of the switches is connected to one of the I/O lines from the 6522 VIAs. The joystick switches are arranged as follows:

Button' Switch 4   Switch 0   TOP (north)

Switch 2———————————————Switch 3

Switch 1

216
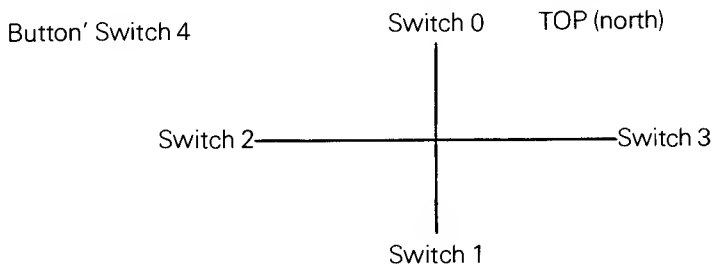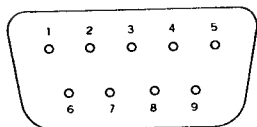
PIN	TYPE	NOTE
1	JOY0	
2	JOY1	
3	JOY2	
4	JOY3	
5	POT Y	
6	LIGHT PEN	
7	+5V	MAX. 100mA
8	GND	
9	POT X	

Fig. 36 — The allocation and function of pins on the Joystick connector.
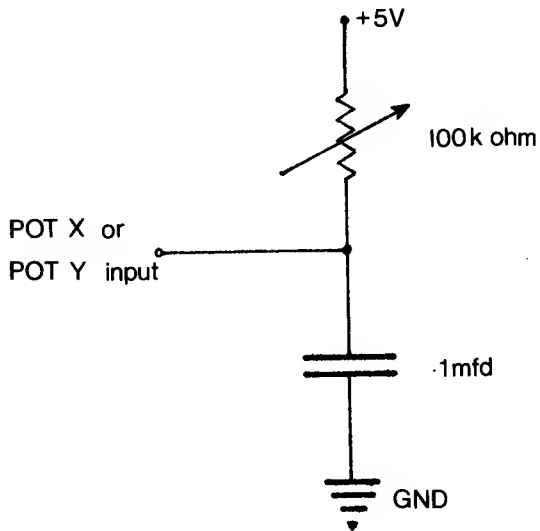
Fig. 37 — Potentiometer joystick circuit.

217

Switches 0, 1 and 2 and the 'Fire button' are connected to lines from VIA No. 1., and switch 3 to a line from VIA No. 2. The VIA memory locations used by the switch joystick are as follows:

Hex	Decimal	Function
$9113	37139	Data Direction Register for Port A VIA No. 1.
$9111	37137	Output Register A
		bit 2 — joystick switch 0
		bit 3 — joystick switch 1
		bit 4 — joystick switch 2
		bit 5 — 'Fire button'
$9122	37154	Data Direction Register for Port B VIA No. 2.
$9120	37152	Output Register B
		bit 7 — joystick switch 3

To read the joystick switch inputs the I/O lines used must first be set into the input mode. Achieved by setting the corresponding bit of the Data direction Register to 0. This poses one problem, the line used for joystick switch 3 is also used for scanning the keyboard. Thus the keyboard can not be used in full at the same time as the switch joystick, and the Data Direction Register should always be restored to normal after the joystick is used. The following program can be used to initialise the Data Direction Registers and input the switch position.

```
10 POKE 37139,0 : POKE 37154,127 : set up DDRs
20 S = PEEK (37137) : input from VIA No. 1.
30 S0 = ((S AND 4) =) : switch 0
40 S1 = ((S AND S) : switch 1
50 S2 = ((S AND 16) = 0) : switch 2
60 F = ((S AND 32) = 0) : 'Fire button'
70 S = PEEK (37152) : input from VIA NO. 2.
80 S3 = — ((S AND 128) = 0) : switch 3
90 POKE 37154,255 : restore keyboard function
```

The variables S0, S1, S2 and S3 will normally be 0 but if the joystick handle is pointed in that direction their value will be either 1 or —1. If the 'Fire button' is pressed then the variable F will have a value of 1,

otherwise it will be 0. These variables can be used to decode the
the joystick into the following pattern:

TOP

```
        0
      7   1
    6   8   2
      5   3
        4
```

The following program lines will convert the variables S0, S1, S2
and S3 into the values shown in the pattern which correspond to the
handle position and store in variable P:

```
100 DATA 7, 0, 1, 6, 8, 2, 5, 4, 3 : data for joystick pattern
110 FOR I = 0 TO 2
120 FOR J = 0 TO 2
130 READ JS (J, I) : put joystick pattern into array
140 NEXT J, I
150 X = 1+(S2 + S3) : Y = 1+(S0 + S1)
160 P = JS(X, Y) : set P to joystick pattern value
```

Potentiometer Joystick

A potentiometer joystick consists of two potentiometers mounted
at right angles to each other in a mechanism which allows the joystick
when moved to change the wiper position on one or both
potentiometers. One potentiometer registers the joystick movement
in the X axis, the other in the Y axis. The rotational movement of each
potentiometer is divided by the computer into 255 divisions. With the
joystick centered vertically the X and Y potentiometers will each have
a value of 128. The position of the joystick can thus be mapped in
terms of graph co-ordinates, thus:

219

X AXIS POTENTIOMETER

255 (top)

180

200*

128

Y AXIS 255 ——————————— * ————————— 0 (right)
POTENTIOMETER   128

128

0 (bottom)
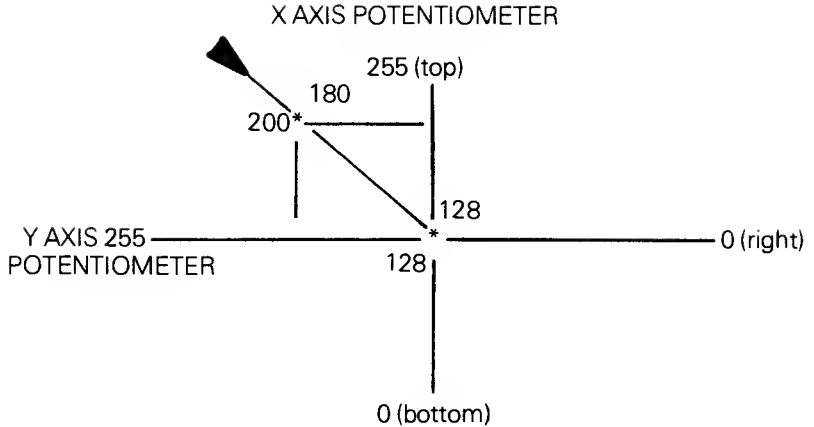
The two potentiometers are connected together with a small amount of additional circuitry to two special inputs on the 6561 VIC chip, their pin assignments on the output connector are shown in Figure 36, The input to the 6561 is used to convert the potentiometer position into a microprocessor readable 8-bit number. This is accomplished by a simple RC time constant integration technique. The potentiometer is used to charge an external capacitor connected to one of the pot pins and ground. This simple circuit is shown in Figure 37.

The 6561 converts the potentiometer position into a value which the processor can read by accessing one of the two potentiometer registers, the memory location of these two registers is;

Hex $9008   decimal 36872 — digitised value of POT X
Hex $9009   decimal 36873 — digitised value of POT Y

The value stored in these two registers can be accessed simply using PEEK or LDA commands.

THE MEMORY EXPANSION CONNECTOR

The memory expansion connector allows additional memory or I/O to be added to the VIC. The 44 line connector gives external equipment access to the VIC system data bus and address bus plus the necessary control lines. These connections are shown in Figure 38. The connector required to attach equipment to the expansion connector is a 44 pin (22/22) male edge connector with a .156 inch connector separation (a double sided etched PC board can be used). The user must exercise great care when interfacing equipment to these lines, since they are not buffered and any malfunction of the external equipment may damage the VIC. The memory expansion port lines can be divided into five groups:

Data Bus — the eight data lines used to transfer data between processor and memory.

Address Bus — the fourteen least significant address lines are available, they allow any memory location in an 8K block to be accessed by the processor. Which of the 8 memory blocks is accessed depends on the block select lines.

Control Bus — the six control lines govern system clock, IRQs, Reset, and R/W select.

Block Select — there are nine block select lines, these are generated by partly decoding the most significant address lines. They are used to select the block of memory or I/O addressed by the I/O bus.

Power Lines — power output is available at +5 volts and Ground, the power rating is approximately 750ma.

The signals available on the memory expansion connector are as follows:

Name	Pin No.	Description
GND	1	System Ground
GD0	2	Data bus line0
CD1	3	Data bus line 1
CD2	4	Data bus line2
CD3	5	Data bus line 3
CD4	6	Data bus line 4
CD5	7	Data bus line 5

MEMORY EXPANSION
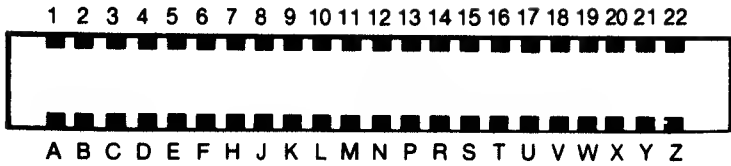
PIN #	TYPE
1	GND
2	CD∅
3	CD1
4	CD2
5	CD3
6	CD4
7	CD5
8	CD6
9	CD7
1∅	$\overline{BLK1}$
11	$\overline{BLK2}$

PIN #	TYPE
12	$\overline{BLK3}$
13	$\overline{BLK5}$
14	$\overline{RAM1}$
15	$\overline{RAM2}$
16	$\overline{RAM3}$
17	VR/W
18	CR/W
19	$\overline{IRQ}$
2∅	NC
21	+5V
22	GND

PIN #	TYPE
A	GND
B	CA∅
C	CA1
D	CA2
E	CA3
F	CA4
H	CA5
J	CA6
K	CA7
L	CA8
M	CA9

PIN #	TYPE
N	CA1∅
P	CA11
R	CA12
S	CA13
T	I/∅2
U	I/∅3
V	S∅2
W	$\overline{NMI}$
X	$\overline{RESET}$
Y	NC
Z	GND

Fig. 38 — The allocation and function of pins on the Memory Expansion connector.

CD6	8	Data bus line 6
CD7	9	Data bus line 7
BLK1	10	8K decoded RAM/ROM block 1, starting at $2000, (active low).
BLK2	11	8K decoded RAM/ROM block 2, starting at $4000, (active low).
BLK3	12	8K decoded RAM/ROM block 3, starting at $6000, (active low).
BLK5	13	8K decoded ROM block 5, starting at $A000 (active low).
RAM1	14	1K decoded RAM at $0400, (active low).
RAM2	15	1K decoded RAM at $0800, (active low).
RAM3	16	1K decoded RAM at $0C00, (active low).
VR/W	17	Read/Write line from VIC chip, (high = read low = write).
CR/W	18	Read/Write line from CPU. (high = read, low = write).
IRQ	19	6502 IRQ line, (active low).
(NC)	20	
+5v	21	+5 volt power line.
GND	22	System Ground.
GND	A	System Ground.
CA0	B	Address bus line 0
CA1	C	Address bus line 1
CA2	D	Address bus line 2
CA3	E	Address bus line 3
CA4	F	Address bus line 4
CA5	H	Address bus line 5
CA6	J	Address bus line 6
CA7	K	Address bus line 7
CA8	L	Address bus line 8
CA9	M	Address bus line 9
CA10	N	Address bus line 10
CA11	P	Address bus line 11
CA12	R	Address bus line 12
CA13	S	Address bus line 13
I/O2	T	Decoded I/O block 2, starting at $9130
I/O3	U	Decoded I/O block 3, starting at $9140
SO2	V	Phase 2 system clock
NMI	W	6502 NMI line, (active low)
RESET	X	6502 RESET line, (active low)
(NC)	Y	
GND	Z	System ground

THE SERIAL IEEE PORT

This is a very cut-down version of the IEEE-488 port available on PET computers. In the normal IEEE, bus data is transferred in parallel form on eight data lines. In the VIC implementation, it is transferred serially on a single line. The VIC IEEE bus consists of just six lines, three output and three input. The three input lines carry data and control pulses from a communicating device to the VIC, the three output lines have an identical function, and output data from the VIC to the peripheral device. The three lines consist of a serial data line, a clock line to clock pulses off the data line, and a service request or attention command line. The functioning of the serial IEEE port on the VIC is very rudimentary compared to the IEEE implemented on the PET, but is adequate for many applications requiring communications between the VIC, and, either a peripheral device, other VICs or a larger computer. If the full IEEE-488 bus is required then the IEEE-488 expansion module should be used. This is very useful if one wishes to connect the VIC to other IEEE-488 devices, in particular the PET peripherals.

Definition of the IEEE port

An IEEE-488 type port, whether the simple serial port available on the unenhanced VIC, or the full implementation of the expansion module, has considerable advantages over a serial RS232 port or a parallel user port. The advantage is that an IEEE-488 type port is capable of communicating with more than one device connected to a single set of I/O lines. It does this by means of the control lines and a strict protocol of commands between the listening device and the talking device. There are three classes of device which can be attached to the IEE bus, they are:

Controller — one device which controls bus operation
Listener — a device receiving data from the bus
Talker — a device transmitting data onto the bus

With the existing operating system software in the VIC, only the VIC can act as a controller, though it can also act as either a listener or talker. All the peripheral devices can be either listeners or talkers, though only one device at a time may be a talker on the bus, Figure 40 shows how the VIC and peripheral devices communicate via the IEEE bus. The 'controller' as its name implies controls the data transfer along the bus, and determines which devices act as 'listeners' and which device is the 'talker'. It does this by individually addressing each
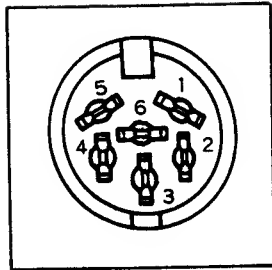
PIN#	TYPE
1	SERIAL SRQ IN
2	GND
3	SERIAL ATN IN/OUT
4	SERIAL CLK IN/OUT
5	SERIAL DATA IN/OUT
6	NC

Fig. 39 — The allocation and function of pins on the IEEE Port connector.
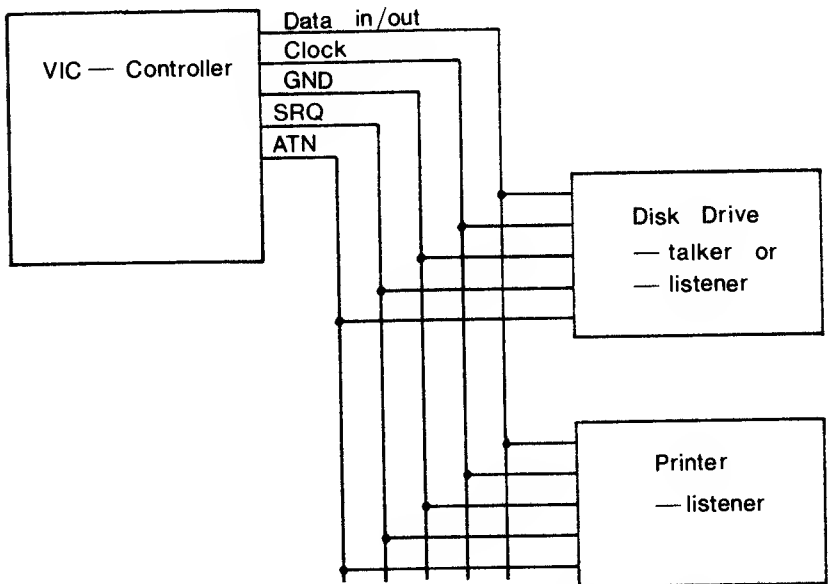
Fig. 40 — The interconnection of devices via the serial IEEE port.

225

device and sending it a set of commands, these set the device as either a 'listener' or 'talker' and in addition can control other functions of the device. Each device has its own unique address code which is usually defined in the devices electronic hardware, the device numbers can be any value between 4 and 30. Having set up the direction of a data transfer of each device and its mode of operation the 'controller' sends out a command to initiate data transfer. When that data transfer is completed the 'talker' sends a command to the 'controller' which then sends commands to the bus devices which disables them as either 'talkers' or 'listeners'.

The serial IEEE port connections.
   The six I/O lines of the VIC serial IEEE port are derived from the two 6522 peripheral I/O chips. The following table shows the derivation of each line.

VIA No.	line No.	Line function
VIA 1	PA1	Serial data in
VIA 2	CB2	Serial data out
VIA 1	PA0	Serial clock in
VIA 2	CA2	Serial clock out
VIA 1	PA7	Serial ATN out
VIA 2	CB1	Serial SRQ in

The output connector and the circuit used to input and output these lines is shown in Figure 39. It should be noted that the 'ATN in' line is not implemented and is simply connected to pin 9 of the user port connector. If 'ATN in' is required then the user should connect pin 9 to one of the unused user port handshake lines and write the appropriate software to handle an 'ATN in' input.

Using the Serial IEEE Port.
   Whether the IEEE port on the VIC is the simple serial port on the unexpanded machine or the full implementation using the external IEEE-488 module the Basic command syntax is identical. The differences lie in the way the data is transmitted. The commands in the following synopsis can be used with either mode of IEEE data transmission, providing that the device or devices communicating with the VIC over the bus are capable of the selected type of communications.

Opening an IEEE channel.

Basic Syntax: OPEN   If, d, sa, "fn"

If — Normal logical file ID (1-255).

d — Device number (4-30). This selects the device to receive this command sequence. A different device number is allocated to each device communicating with the VIC via the IEEE bus.

sa — Secondary address (0-31). This code value is used to determine the operating mode of an intelligent peripheral. By changing the secondary address the operating characteristics of the device can be changed, the value used and its operation will be unique to the addressed device.

"fn" — File name string. The file name field is an extension of the secondary address and is principally used when communicating with storage devices such as tape and disk drives. The file name field can be either a string, or string variables up to 128 characters long, and is used to specify a data item or a file name. The use of a file name and the syntax used to construct the string is dependant on the device addressed.

Function of the OPEN Command
The OPEN command selects a device that has a value between 4 and 30 and the operating system assumes that the device is an IEEE device. If no file name or secondary address is specified then nothing is communicated to the peripheral devices from the VIC 'controller'. The operating system takes the variables in the OPEN command and stores them in the file tables. However, if a file name is specified the operating system sends a 'listen attention' sequence to the device specified in the OPEN command. The secondary address is also transmitted with the file name as the hexadecimal or of $F0 and the secondary address specified in the OPEN command. The VIC operating system allows up to ten logical files to be opened at any one time.

Machine Code Entry Points for Serial IEEE.

Set logical, first, and second address	— $FFBA
Set file name	— $FFBD
Open command routine	— $FFC0

DATA OUT FROM VIC

10μs

Data Valid

DATA IN TO VIC

EOI flag pulse

* - Indefinite length

32μs

255μs

CLOCK OUT FROM VIC

bit 0
(least significant)

24μs

90μs

bit 1

bit 7
(most significant)
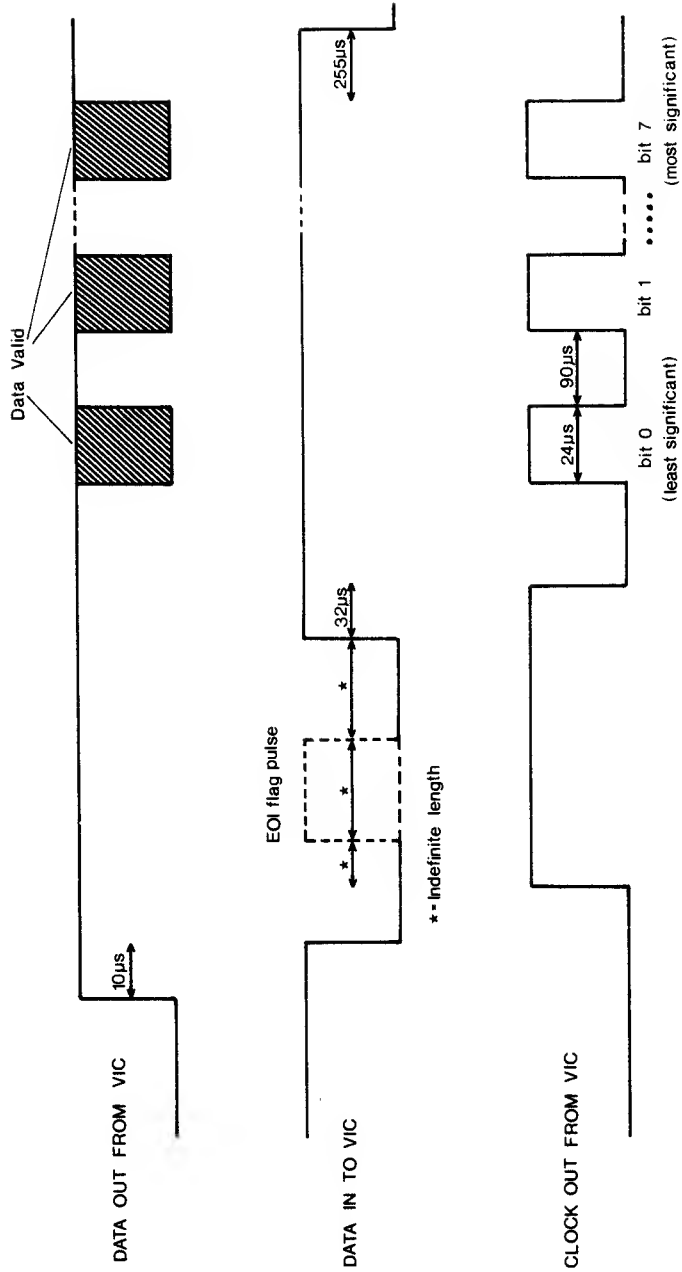
Fig. 41 — Waveform of data and control lines on the VIC IEEE port.

228

Receiving Data From an IEEE Channel

Basic Syntax: either   INPUT #   lf, V
           or   GET #   lf, V

lf — Logical file ID specified for the device in the OPEN command.

V — Input data stored in variable V or V$.

The INPUT #   command accepts characters from the peripheral and builds them up into the variable V. This continues until the delimiter character is received. The delimiter character is a carriage return (CHR$ 13) and marks the end of the input. The variable string is built up in the Basic input buffer, this has a maximum length of 88 characters, an input string should therefore not exceed 88 characters between carriage returns. The GET #   command is used to get a single character from the bus, no delimiter is needed. The GET command is also not subject to the 88 character buffer limitation and can be used to input or output string greater than 88 characters.

IEEE Device Input Sequence and Function.
All INPUT #   and GET #   commands go through the same sequence.
The IEEE initiation routine is first called, this sends a 'Talk Attention' sequence to the device, followed by the secondary address specified for that logical file in the OPEN command. At the end of the 'Attention' sequence the VIC establishes itself in the 'listener' mode and waits for a signal from the addressed device, indicating that a single character has been received. If this signal is not received within 64ms then an error is generated and the correct code stored in the status byte, variable ST. If the signal is received within the timeout period then control is passed to the IEEE input routine. The IEEE input routine gets a single character from the bus using the clock line to clock each bit off the serial data input line. If during the course of inputting data an EOI signal is received then the IEEE routine will set the EOI status flag, this indicates that the next byte is the last byte. This calls the termination 'Untalk' routine which returns command to the keyboard and sends an 'Untalk' command to the IEEE bus thereby freeing the bus for the next command. Figure 41 and 42 show the flow of data between the VIC and the serial IEEE bus devices with the relevent pulse sequence and timings.

229

Machine Code Entry Points For Serial IEEE Input Routines:
  Command serial bus device to Talk — $FFB4
  Send secondary address after Talk — $FF96
  Input byte from serial IEEE port — $FFA5
  Command serial bus to Untalk — $FFAB
  Set timeout on IEEE bus — $FFA2

Transmitting Data to an IEEE Channel.

Basic Syntax: PRINT #   lf, V

  lf — Logical file ID specified for the device in the OPEN command.

  V — Output data stored in variable V or V$.

IEEE Output Sequence and Function.
    The PRINT # , command first calls a routine which sends a 'Listen Attention' command to the addressed device on the bus, this sets that device as a 'listener'. This is followed by the secondary address byte specified for that logical file in the OPEN command. The VIC expects a response signal from the listening device within 256 µs otherwise a device not present error is signalled. The IEEE output routine then transmits the data in the variable bit by bit down the serial output line together with synchronising clock pulses. The output data is stored in the Basic buffer prior to transmission, and it is from here that the output routine accesses each byte. When the last byte of data to be transmitted is reached the VIC sends an EOI signal to the listener to warn the listening device that transmission is about to end. Having transmitted this last byte the VIC sends an 'Unlisten' command to the bus and restores output to the screen. This frees the bus for the next operation. Figures 41 and 42 show the flow of data between the VIC and a serial IEEE peripheral device, together with the pulse sequences and timings.

Machine Code Entry Points for Serial IEEE Output Routines

    Command serial bus device to Listen — $FFB1
    Send secondary address after Listen — $FF93
    Output byte to serial IEEE port — $FFA8
    Command serial bus to Unlisten — $FFAE
    Set timout on IEEE bus — $FFA2

Closing an IEEE Channel.

Basic Syntax: CLOSE lf

lf — Logical file ID specified in OPEN command.

IEEE Named Device Closure.
When an IEEE file which was opened with a file name is closed a special command sequence is generated. This command sequence sends the secondary address from the OPEN command ORed with hexadecimal $E0 to the device specified. This allows special file closure commands to be sent to intelligent peripherals.

Machine Code Entry Point for Serial IEEE Close Routine.

Close named IEEE device — $FFC3

Other IEEE Commands
There are three special IEEE commands, they are: LOAD, SAVE, and CMD. The first two are concerned with the transfer of programs between the VIC and a peripheral device on the IEEE bus. The last command, CMD, is a special form of the PRINT No., command. All three commands should be preceded and followed by the OPEN and CLOSE command specifying the device number to be accessed. The function and syntax of these three commands is as follows:

Load program from IEEE device.

Basic Syntax: LOAD fn, d, sa

fn — File name of program to be loaded into the VIC memory, may contain optional commands to the addressed device such as disk drive number. The file name and optional device directive should be enclosed in quotes.

d — Device number defined in the OPEN command.
sa — Optional secondary address command.

The first two bytes of data retreived in a LOAD command contain the starting address of the program.

231

Save program on IEEE device.

Basic Syntax: SAVE fn, d, sa

fn — File name of program to be saved on peripheral device. The file name should be enclosed in quotes and may contain an optional command to the addressed device eg: disk drive number.

d — Device number defined in the OPEN command.
sa — Optional secondary address command.

The starting address of the program in VIC memory is transmitted in the first two bytes of data.

The CMD command.

Basic Syntax: CMD lf, V

lf — Logical file ID specified for the device in the OPEN command.
V — Output data stored in variable V or V$.

The CMD command is virtually identical to the PRINT No., command, except that at the end of data transfer the unlisten routine is not called, thereby leaving the device to be commanded by a CMD as the primary output device for Basic. PRINT or LIST commands are then directed to this device rather than to the video screen. The most frequent use of CMD is in obtaining printed program listings. The CMD command is terminated by a PRINT No., command being executed.

Important memory locations used by the VIC serial IEEE port.

$90	— The I/O status flag
$94	— IEEE buffered character flag
$95	— IEEE buffered character
$97	— Temp for IEEE input
$98	— Pointer to file table
$99	— Input device No.
$9A	— Output CMD device
$A3	— Serial bit cont/EOI flag
$A4	— Cycle counter for serial I/O
$B7	— Length of current file name string
$B8	— Current logical file number

TALKER

Flowchart beginning at connector **A**.

- **Dout → high** — Dout
- **Din → low within 256us** — Din → (no) → **Timeout Error / Device not Present**
- **Cout → high** — Cout
- **Din = high?** — Din (no loops back)
  - yes ↓
- **Last byte before 'Untalk' & 'Unlisten'? EOI set**
  - yes → **Cout → low** — Cout → **Din = low?** — Din (no loops) → yes
  - no ↓
- **Din = high?** — Din (no loops back)
  - yes ↓
- **Cout → low** — Cout
- **Output data byte** — Dout
- **Din = low?** — Din (no loops back)
  - yes ↓
- connector **A**

233
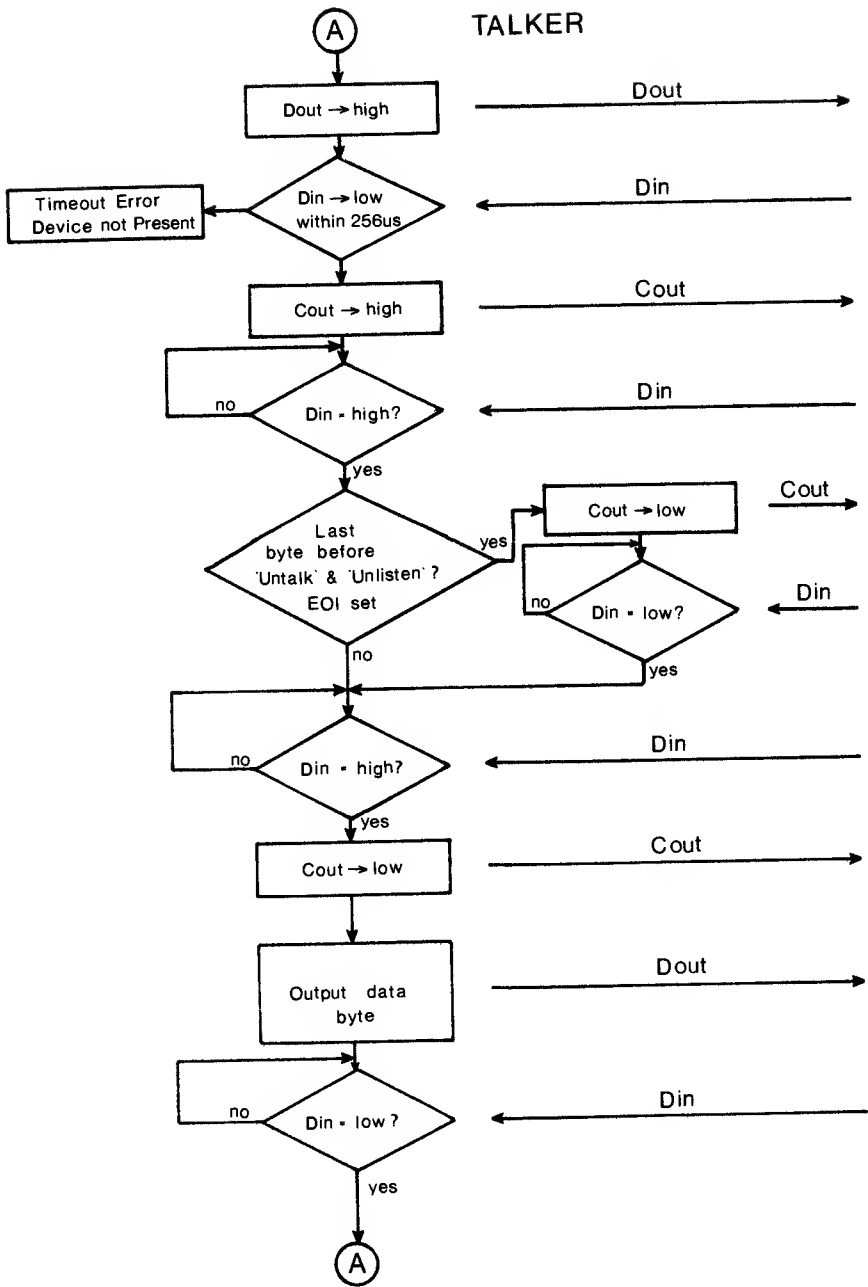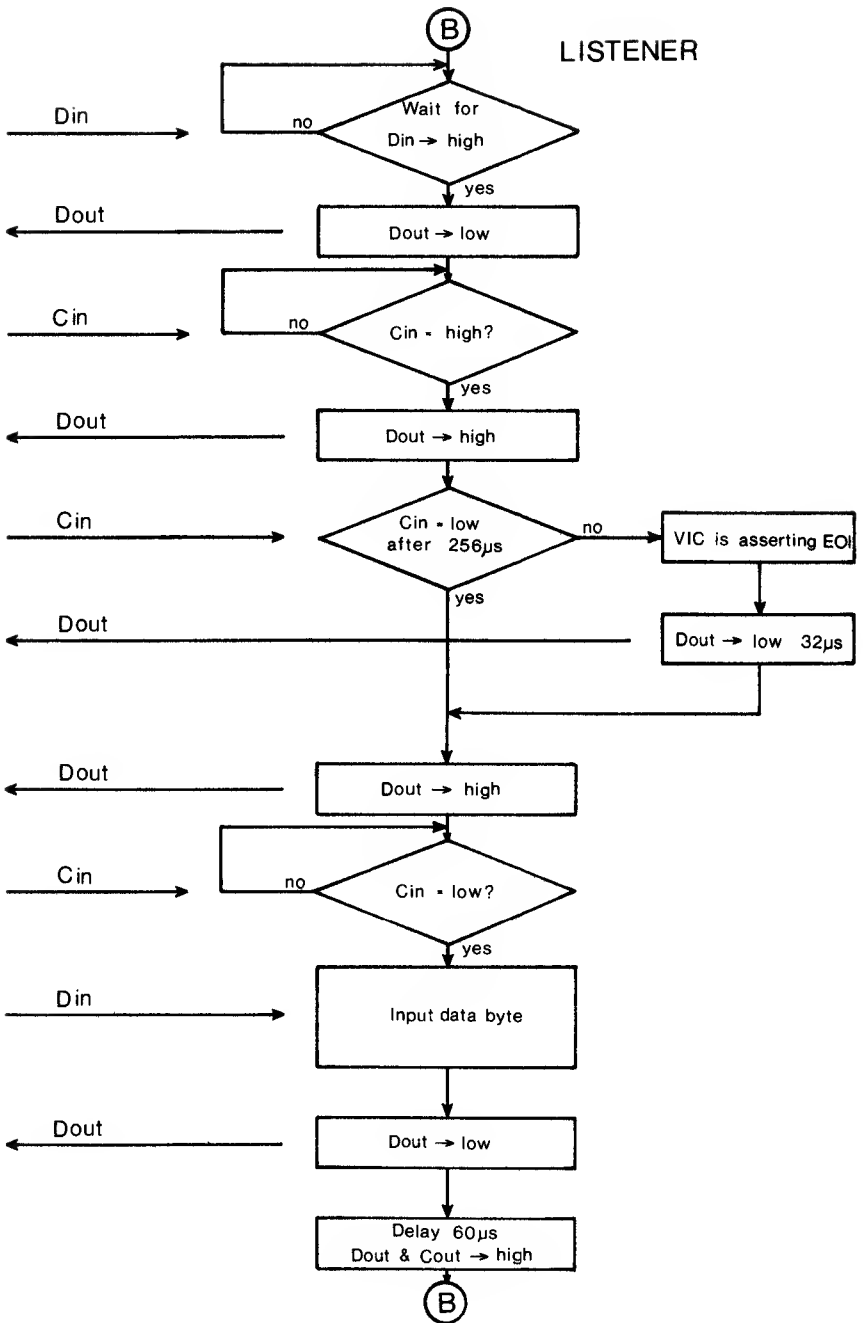
Fig. 42 — Flow diagrams of data input and output sequence in IEEE port communications.

```
$B9    — Current secondary address
$BA    — Current device number
$BB    — Address of current file name string
$0200  — 88 byte Basic input buffer
$0259  — Logical file number table, 10 bytes
$0263  — Device number table, 10 bytes
$026D  — Secondary address table, 10 bytes
$0285  — IEEE timeout flag
```

System subroutine locations for serial IEEE

```
$E4A0  — Set data line high
$E4A9  — Set data line low
$E4B2  — Debounce PIA and shift clock to carry
$EE14  — Command serial bus device to talk
$EE17  — Command serial bus device to listen
$EE40  — Output a byte from serial bus
$EE6F  — Set to send data
$EEC0  — Send secondary address after listen
$EEC5  — Release attention after listen
$EECE  — Talk second address
$EED3  — Talk attention
$EEE4  — Buffered output to serial bus
$EEF6  — Send untalk command to serial bus
$EF04  — Send unlisten command to serial bus
$EF19  — Input a byte from serial bus
$EF84  — Set clock line high
$EF8D  — Set clock line low
$EF96  — Delay 1ms
```

Vector jump addresses for serial IEEE

```
$FF93  — Send secondary address after listen
$FF96  — Send secondary address after talk
$FFA2  — Set timeout on IEEE bus
$FFA5  — Input byte from serial IEEE port
$FFA8  — Output byte to serial IEEE port
$FFAB  — Command serial bus device to untalk
$FFAE  — Command serial bus device to unlisten
$FFB1  — Command serial bus device to listen
$FFB4  — Command serial bus device to talk
$FFBA  — Set logical, first and second address
```

235

$FFBD — Set file name
$FFC0 — Perform OPEN command
$FFC3 — Perform CLOSE command

Table of CBM Codes

DECIMAL	HEX	ASCII	SCREEN	BASIC	6502	DECIMAL
0	00		@	end-line	BRK	0
1	01		A		ORA(I,X)	1
2	02		B			2
3	03		C			3
4	04		D			4
5	05		E		ORA Z	5
6	06		F		ASL Z	6
7	07		G			7
8	08		H		PHP	8
9	09		I		ORA #	9
10	0A		J		ASL A	10
11	0B		K			11
12	0C		L			12
13	0D	car ret	M		ORA	13
14	0E		N		ASL	14
15	0F		O			15
16	10		P		BPL	16
17	11	cur down	Q		ORA(I),Y	17
18	12	reverse	R			18
19	13	cur home	S			19
20	14	delete	T			20
21	15		U		ORA Z,X	21
22	16		V		ASL Z,X	22
23	17		W			23
24	18		X		CLC	24
25	19		Y		ORA Y	25
26	1A		Z			26
27	1B		[			27
28	1C		\			28
29	1D	cur right	]		ORA X	29
30	1E		↑		ASL X	30
31	1F		←			31
32	20	space	space	space	JSR	32
33	21	!	!	!	AND(I,X)	33
34	22	"	"	"		34
35	23	#	#	#		35
36	24	$	$	$	BIT Z	36
37	25	%	%	%	AND Z	37
38	26	&	&	&	ROL Z	38
39	27	'	'	'		39
40	28	(	(	(	PLP	40
41	29	)	)	)	AND #	41
42	2A	*	*	*	ROL A	42
43	2B	+	+	+		43
44	2C	,	,	,	BIT	44
45	2D	-	-	-	AND	45
46	2E	.	.	.	ROL	46
47	2F	/	/	/		47
48	30	0	0	0	BMI	48
49	31	1	1	1	AND(I),Y	49

237

DECIMAL	HEX	ASCII	SCREEN	BASIC	6502	DECIMAL
50	32	2	2	2		50
51	33	3	3	3		51
52	34	4	4	4		52
53	35	5	5	5	AND Z,X	53
54	36	6	6	6	ROL Z,X	54
55	37	7	7	7		55
56	38	8	8	8	SEC	56
57	39	9	9	9	AND Y	57
58	3A	:	:	:	CLI	58
59	3B	;	;	;		59
60	3C					60
61	3D	=	=	=	AND X	61
62	3E				ROL X	62
63	3F	?	?	?		63
64	40				RTI	64
65	41	A	,a	A	EOR(I,X)	65
66	42	B	,b	B		66
67	43	C	,c	C		67
68	44	D	,d	D		68
69	45	E	,e	E	EOR Z	69
70	46	F	,f	F	LSR Z	70
71	47	G	,g	G		71
72	48	H	,h	H	PHA	72
73	49	I	,i	I	EOR #	73
74	4A	J	,j	J	LSR A	74
75	4B	K	,k	K		75
76	4C	L	,l	L	JMP	76
77	4D	M	,m	M	EOR	77
78	4E	N	,n	N	LSR	78
79	4F	O	,o	O		79
80	50	P	,p	P	BVC	80
81	51	Q	,q	Q	EOR(I),Y	81
82	52	R	,r	R		82
83	53	S	,s	S		83
84	54	T	t,	T		84
85	55	U	u,	U	EOR Z,X	85
86	56	V	,v	V	LSR Z,X	86
87	57	W	,w	W		87
88	58	X	,x	X	CLI	88
89	59	Y	,y	Y	EOR Y	89
90	5A	Z	,z	Z		90
91	5B					91
92	5C					92
93	5D				EOR X	93
94	5E		,		LSR X	94
95	5F		,			95
96	60				RTS	96
97	61				ADC(I,X)	97
98	62					98
99	63					99

DECIMAL	HEX	ASCII	SCREEN	BASIC	6502	DECIMAL
100	64					100
101	65				ADC Z	101
102	66				RCR Z	102
103	67					103
104	68				PLA	104
105	69				ADC #	105
106	6A				RCR A	106
107	6B					107
108	6C				JMP(I)	108
109	6D				ADC	109
110	6E				ROR	110
111	6F					111
112	70				BVS	112
113	71				ADC(I),Y	113
114	72					114
115	73					115
116	74					116
117	75				ADC Z,X	117
118	76				ROR Z,X	118
119	77					119
120	78				SEI	120
121	79				ADC Y	121
122	7A					122
123	7B					123
124	7C					124
125	7D				ADC X	125
126	7E				ROR X	126
127	7F					127
128	80		r-@	END		128
129	81		r-A	FOR	STA(I,X)	129
130	82		r-8	NEXT		130
131	83		r-C	DATA		131
132	84		r-D	INPUT #	STY Z	132
133	85		r-E	INPUT	STA Z	133
134	86		r-F	DIM	STX Z	134
135	87		r-G	READ		135
136	88		r-H	LET	DEY	136
137	89		r-I	GOTO		137
138	8A		r-J	RUN	TXA	138
139	88		r-K	IF		139
140	8C		r-L	RESTORE	STY	140
141	8D	car ret	r-M	GOSUB	STA	141
142	8E		r-N	RETURN	STX	142
143	8F		r-0	REM		143
144	90		r-P	STOP	8CC	144
145	91	cur up	r-Q	ON	STA(I),Y	145
146	92	rvs off	r-R	WAIT		146
147	93	clear	r-S	LOAD		147
148	94	insert	r-T	SAVE	STY Z,X	148
149	95		r-U	VERIFY	STA Z,X	149

239

DECIMAL	HEX	ASCII	SCREEN	BASIC	6502	DECIMAL
150	96		r-V	DEF	STX Z,Y	150
151	97		r-W	POKE		151
152	98		r-X	PRINT #	TYA	152
153	99		r-Y	PRINT	STA Y	153
154	9A		r-Z	CONT	TXS	154
155	9B		r-[	LIST		155
156	9C		r-\	CLR		156
157	9D	cur left	r-]	CMD	STA X	157
158	9E		r-↑	SYS		158
159	9F		r- ←	OPEN		159
160	A0		■	CLOSE	LDY #	160
161	A1		r-!	GET	LDA(I,X)	161
162	A2		r-"	NEW	LDX #	162
163	A3		r-#	TAB(		163
164	A4		r-$	TO	LDY Z	164
165	A5		r-%	FN	LDA Z	165
166	A6		r-&	SPC(	LDX Z	166
167	A7		r-'	THEN		167
168	A8		r-(	NOT	TAY	168
169	A9	, ▨	r-)	STEP	LDA #	169
170	AA		r-*	+	TAX	170
171	AB		r-+	-		171
172	AC		r-,	*	LDY	172
173	AD		r--	/	LDA	173
174	AE		r-.		LDX	174
175	AF		r-/	AND		175
176	B0		r-0	OR	BCS	176
177	B1		r-1		LDA(I),Y	177
178	B2		r-2	=		178
179	B3		r-3			179
180	B4		r-4	SGN	LDY Z,X	180
181	B5		r-5	INT	LDA Z,X	181
182	B6		r-6	ABS	LDX Z,Y	182
183	B7		r-7	USR		183
184	B8		r-8	FRE	CLV	184
185	B9		r-9	POS	LDA Y	185
186	BA	, ▱	r-:	SQR	TSX	186
187	BB		r-;	RND		187
188	BC		r-	LOG	LDY X	188
189	BD		r-=	EXP	LDA X	189
190	BE		r-	COS	LDX Y	190
191	BF		r-?	SIN		191
192	C0			TAN	CPY #	192
193	C1	, a		ATN	CMP(I),X	193
194	C2	, b		PEEK		194
195	C3	, c		LEN		195
196	C4	, d		STR$	CPY Z	196
197	C5	, e		VAL	CMP Z	197
198	C6	, f		ASC	DEC Z	198
199	C7	, g		CHR$		199

240

DECIMAL	HEX	ASCII	SCREEN	BASIC	6502	DECIMAL
200	C8	, h		LEFT$	INY	200
201	C9	, i		RIGHT$	CMP #	201
202	CA	, j		MID$	DEX	202
203	CB	, k				203
204	CC	, l			CYP	204
205	CD	, m			CMP	205
206	CE	, n			DEC	206
207	CF	, o				207
208	DO	, p			BNE	208
209	D1	, q			CMP(I),Y	209
210	D2	, r				210
211	D3	, s				211
212	D4	, t				212
213	D5	, u	Reverse of ASCII		CMP Z,X	213
214	D6	, v			DEC Z,X	214
215	D7	, w				215
216	D8	, x			CLD	216
217	D9	, y			CMP Y	217
218	DA	, z				218
219	DB					219
220	DC					220
221	DD				CMP X	221
222	DE	π ,			DEC X	222
223	DF	,				223
224	EO				CPX #	224
225	E1				SBC(I),X	225
226	E2					226
227	E3					227
228	E4				CPX Z	228
229	E5				SBC Z	229
230	E6				INC Z	230
231	E7					231
232	E8				INX	232
233	E9				S8C #	233
234	EA				NOP	234
235	EB					235
236	EC				CPX	236
237	ED				S8C	237
238	EE				INC	238
239	EF					239
240	FO				8EQ	240
241	F1				S8C(I),Y	241
242	F2					242
243	F3					243
244	F4					244
245	F5				S8C Z,X	245
246	F6				INC Z,X	246
247	F7					247
248	F8				SED	248
249	F9				SBC Y	249

DECIMAL	HEX	ASCII	SCREEN	BASIC	6502	DECIMAL
250	FA					250
251	FB					251
252	FC					252
253	FD				SBC X	253
254	FE				INC X	254
255	FF			π		255

LOC	CODE	LINE

```
9600                ; ENTRY CODE
9600                ;
9600                        *=$A000
A000   09 A0            .WOR START
A002   2B A0            .WOR PANIC
A004   41 30            .BYT 'A0',$C3,$C2,$CD ;AOCBM
A006   C3
A007   C2
A008   CD
A009                ;
A009                ; START - SET UP OP SYSTEM
A009                ;
A009   20 3F A0     START  JSR MYINIT      ;GO RESET SYSTEM
A00C                ;
A00C                ; START - SET UP BASIC SYSTEM
A00C                ;
A00C   20 13 A0            JSR INITVN      ;INITILIZE VECTORS
A00F   58                  CLI
A010   4C 7B E3            JMP INITNV      ;INITILIZE REST
A013                ;
A013                ; INITVN ;INITILIZE THE VECTORS MY WAY
A013                ;
A013   A2 0B        INITVN LDX #ITEND-ITBGN-1
A015   BD 1F A0     INITVL LDA ITBGN,X
A018   9D 00 03            STA IERROR,X
A01B   CA                  DEX
A01C   10 F7               BPL INITVL
A01E   60                  RTS
A01F                ;
A01F   3A C4        ITBGN   WOR NERROR,NMAIN,CNCHST,LISTER,
A021   83 C4                            BYEBYE,EVALMY
A023   81 A0
A025   E3 A0
A027   16 A1
A029   3C A1
A02B                ITEND
A02B                ;
A02B                ; PANIC - USE THE OLD RETURN
A02B                ;
A02B   2C 11 91     PANIC  BIT D1ORAH       ;CLR NMI REG
A02E   20 34 F7            JSR UDTIM        ;CHECK FOR STOP KEY
A031   20 E1 FF            JSR STOP
A034   D0 06               BNE PANIC1
A036   20 42 A0            JSR MYREIT       ;RESTORE MY I/O SYSTEM
A039   6C 02 C0            JMP ($C002)      ;RETURN TO BASIC
```

```
A03C    4C 56 FF    PANIC1 JMP PREND        ;DO AN EXIT
A03F                ;
A03F                ; MYINIT - MASTER SET UP CODE
A03F                ; MYREIT - MASTER RESTORE CODE
A03F                ;
A03F    20 8D FD    MYINIT JSR RAMTAS        ;GO TEST RAM
A042    20 8A FF    MYREIT JSR MOVOSI        ;MOVE OS VECTORS
A045    20 F9 FD           JSR IOINIT
A048    20 18 E5           JSR CINT          ;GO INIT SCREEN
A04B    A9 B9              LDA #<TIMB        ;RESET BREAK VECTOR
A04D    8D 16 03           STA CBINV
A050    A9 A2              LDA #>TIMB
A052    8D 17 03           STA CBINV+1
A055    60                 RTS


A056                ; DLIST - HOLDS DISPATCH LOCATIONS
A056                ;   STARTING VALUE @314
A056                DLIST
A056    AD A2              .WOR MONTOR-1    ;MONITOR
A058    64 A1              .WOR SOUND-1
A05A    DF A1              .WOR SETPLT-1    ;SETPLOT
A05C    46 A2              .WOR PLOT-1
A05E                LASTST =@317            ;LAST STATEMENT
A05E    92 A1              .WOR PDL-1            TOKEN VALUE
A060    A3 A1              .WOR JOY-1
A062                LASTFN =@321            ;LAST FUNCTION
                                                TOKEN VALUE


A062                ; LIST - HOLDS ASCII TOKEN TABLES
A062                ;
A062                LIST
A062    4D 41              .BYT 'MACHIN',$C5 ; MACHINE
A068    C5
A069    53 4F              .BYT 'SOUND',$A8 ; SOUND(
A06E    A8
A06F    53 45              .BYT 'SETPLO',$D4 ;SETPLOT
A075    D4
A076    50 4C 4F           .BYT 'PLO',$D4   ;PLOT X,Y
A079    D4
A07A    50 44              .BYT 'PD',$CC    ; PDL
A07C    CC
A07D    4A 4F              .BYT 'JO',$D9    ; JOY
A07F    D9
A080    00                 .BYT $00         ;END OF LIST TABLE
```

```
LOC    CODE          LINE

                     ; NEW CRUNCH ROUTINE
A081                 ;   COME TO HERE ON INDIRECT
A081                 ;   TOKEN CHAR LIST MUST BE <255 CHARS
A081                 ;   COMMAND BUFFER LINE MUST BE <255 CHARS
A081                 ;
A081   20 7C C5      CNCHST JSR NCRNCH      ; GO TOKENIZE ALL OLD SYMBOLS
A084   A0 05                LDY #5          ; SET UP TO TOKENIZE ALL NEW
A086                                                              SYMBOLS
A086                 ;
A086   B9 FB 01      LOOPOT LDA BUF-5, Y    ; GET DATA BYTE
A089   F0 57                BEQ CNCHRT      ; IF ZERO THEN EXIT-RETURN
A08B   C9 22                CMP #'"         ; CHECK FOR QUOTE CASE
A08D   F0 47                BEQ LOOPQT      ; YES...GOTO QUOTE LOOP
A08F                 ;
A08F   C9 41                CMP #'A         ; CHECK IF IN ALPHA RANGE...
A091   90 40                BCC LOOPBK      ; NO...BELOW
A093   C9 5B                CMP #'[         ; 'Z+1
A095   B0 3C                BCS LOOPBK      ; NO...ABOVE
A097                 ;
A097                 ; TOKENIZE IF IN TABLES
A097                 ;
A097   84 B1                STY TEMP        ; HOLD OLD .Y VALUE
A099   A2 00                LDX #0
A09B   86 0B                STX COUNT
A09D                 ;
A09D   38           LOOPIN SEC              ; FIND TOKEN
A09E   FD 62 A0             SBC LIST, X
A0A1   F0 13                BEQ NEXT        ; A MATCHED CHAR
A0A3   C9 80                CMP #128
A0A5   F0 16                BEQ DONE        ; A MATCHED TOKEN
A0A7                 ;
A0A7   BD 62 A0      LOOPNO LDA LIST, X     ; NO MATCH LOOP
A0AA   F0 27                BEQ LOOPBK      ; AT END OF LIST, DISCARD CHAR
A0AC   30 03                BMI CONTLP      ; AT END OF TOKEN, GO COMPARE
A0AE   E8                   INX                                TO NEXT
A0AF   D0 F6                BNE LOOPNO      ; JMP
A0B1                 ;
A0B1   E6 0B        CONTLP INC COUNT        ; INC TOKEN.COUNT
A0B3   A4 B1                LDY TEMP        ; POINT TO BEGINNING OF CHECK
A0B5   A9                   .BYT $A9        ; SKIP 1 (LDA #)
A0B6   C8           NEXT   INY              ; GET NEXT CHAR
A0B7   B9 FB 01             .LDA BUF-5, Y
A0BA   E8                   INX             ; NEXT IN LIST
A0BB   D0 E0                BNE LOOPIN
A0BD                 ;
A0BD                 ; DONE - STORE TOKEN AND COMPACT
A0BD                 ;
A0BD   A6 B1        DONE   LDX TEMP         ; GET OLD POSITION
A0BF   A5 0B                LDA COUNT       ; GET TOKEN VALUE
A0C1   18                   CLC
A0C2   69 CC                ADC #@314       ; LAST BASIC TOKEN+VALUE
A0C4   9D FB 01             STA BUF-5, X    ; PUT INTO POSITION
A0C7                 ;
A0C7   C8           LOOPC  INY              ; CRUNCH COMMAND STRING DOWN
A0C8   E8                   INX
A0C9   B9 FB 01             LDA BUF-5, Y
```

```
LOC    CODE            LINE


AOCC   9D FB 01            STA BUF-5, X
AOCF   DO F6               BNE LOOPC       ; UNTILL WE HIT ENDING ZERO
AOD1   A4 B1               LDY TEMP        ; GET POSISTION, AND CONTINUE
AOD3
AOD3   CB          LOOPBK  INY
AOD4   DO BO               BNE LOOPOT
AOD6
AOD6               ; QUOTE LOOP
AOD6               ;
AOD6   CB          LOOPQT  INY
AOD7   B9 FB 01            LDA BUF-5, Y
AODA   FO 06               BEQ CNCHRT      ; IF ZERO THEN AT END OF LINE
AODC   C9 22               CMP #'"
AODE   DO F6               BNE LOOPQT
AOEO   FO F1               BEQ LOOPBK
AOE2               ;
AOE2   60          CNCHRT  RTS             ; MASTER EXIT

AOE3               ; LISTER - NEW LIST ROUTINE
AOE3               ;   VECTOR QPLOP TO THIS ROUTINE
AOE3               ;
AOE3   08          LISTER  PHP             ; SAVE FOR EXIT
AOE4   C9 FF               CMP #255        ; IF PI THEN EXIT
AOE6   FO 2A               BEQ LEXIT
AOEB   24 OF               BIT DORES       ; IF QUOTES ON THEN EXIT
AOEA   30 26               BMI LEXIT
AOEC   C9 CC               CMP #@314       ; IF NOT IN RANGE THEN EXIT
AOEE   90 22               BCC LEXIT
AOFO   28                  PLP             ; TOSS STACK WILL USE DIFFERENT
AOF1               ;                                        RETURN
AOF1               ; PRINT TOKEN IN LIST
AOF1               ;
AOF1   38                  SEC
AOF2   E9 CB               SBC #@313       ; GET INDEX
AOF4   AA                  TAX
AOF5   B4 49               STY LSTPNT      ; SAVE . Y
AOF7   AO FF               LDY #255
AOF9               ;
AOF9   CA          RESLP1  DEX             ; LOOP UNTILL TOKEN FOUND
AOFA   FO 08               BEQ RESPRT      ; FOUND... PRINT IT
AOFC               ;
AOFC   CB          RESLP2  INY             ; LOOP UNTILL NEXT IN LIST
AOFD   B9 62 AO            LDA LIST, Y                       FOUND
A100   10 FA               BPL RESLP2
A102   30 F5               BMI RESLP1      ; END OF TOKEN
A104               ;
A104   CB          RESPRT  INY             ; PRINT OUT TOKEN-LIST
A105   B9 62 AO            LDA LIST, Y
A108   30 05               BMI RESEXT      ; ALL DONE
A10A   20 D2 FF            JSR BSOUT       ; OUTPUT THE CHAR
A10D   DO F5               BNE RESPRT
A1OF               .
A1OF   4C EF C6    RESEXT  JMP PRIT4       ; GO BACK TO BASIC
A112               ;
A112   28          LEXIT   PLP             ; RESTORE STATUS
A113   4C 1A C7            JMP NQPLOP      ; GO BACK TO NORMAL LIST
```

246

```
LOC     CODE        LINE

A116                    ; BYEBYE - THIS IS THE NEW COMMAND DISPATCHER
A116                    ;
A116    20 73 00    BYEBYE JSR CHRGET       ;GET THE NEXT CHARACTER
A119    C9 CC              CMP #@314        ;CHECK TO SEE IF IN OUR LIST
A11B    90 19             BCC BYERTS        ;NO...LEAVE
A11D    C9 D0             CMP #LASTST+1     ;CHECK TO SEE IF BEYOND
A11F    B0 15             BCS BYERTS        ;YES...LEAVE
A121    20 27 A1          JSR BYEGO
A124    4C AE C7          JMP NEWSTT        ;RETURN
A127                ;
A127    E9 CB       BYEGO  SBC #@313        ;SUB @314
A129    0A                 ASL A            ;MULT *2
A12A    A8                 TAY              ;GOTO THE ROUTINE
A12B    B9 57 A0           LDA DLIST+1,Y
A12E    48                 PHA
A12F    B9 56 A0           LDA DLIST,Y
A132    48                 PHA
A133    4C 73 00           JMP CHRGET       ;SAME DISPATCH AS GONE
A136                ;
A136    20 79 00    BYERTS JSR CHRGOT       ;RESTORE POINTERS
A139    4C E7 C7           JMP NCONE1


A13C    A9 00       EVALMY LDA #0           ;COPY FROM EVAL
A13E    85 0D              STA VALTYP
A140    20 73 00           JSR CHRGET
A143    C9 D0             CMP #LASTST+1     ;IS IT IN RANGE?
A145    90 13             BCC EVALLV        ;NO.
A147    C9 D2             CMP #LASTFN+1
A149    B0 0F             BCS EVALLV        ;NO...
A14B    E9 CB             SBC #@313         ;SUB @314
A14D    0A                ASL A             ;MUL*2
A14E    A8                TAY
A14F    B9 57 A0          LDA DLIST+1,Y
A152    48                PHA
A153    B9 56 A0          LDA DLIST,Y
A156    48                PHA
A157    4C 73 00          JMP CHRGET
A15A                ;
A15A    A5 7A       EVALLV LDA TXTPTR       ;BACK UP TXTPTR
A15C    D0 02             BNE EVALRT
A15E    C6 7B             DEC TXTPTR+1
A160    C6 7A       EVALRT DEC TXTPTR
A162    4C 86 CE          JMP NEVAL
```

247

```
LOC    CODE        LINE

A165                    ;  SOUND(X1,X2,X3,X4,AM)
A165                    ;
A165   A2 00     SOUND    LDX #0              ; INDEX
A167   8E 00 01  SLOOP    STX FBUFFR          ; IN A TEMP
A16A   20 9E D7           JSR GETBYT
A16D   A8                 TAY                 ;SAVE CHRGOT
A16E   8A                 TXA
A16F   AE 00 01           LDX FBUFFR          ; COUNT
A172   9D 01 01           STA FBUFFR+1,X  ; INDEX INTO TEMP AREA
A175   E8                 INX
A176   E0 06              CPX #6              ;CHECK FOR TOO MANY PRAMS
A178   B0 0B              BCS SDERR           ;YES...TOO MANY
A17A   20 73 00           JSR CHRGET          ;GET NEXT CHAR
A17D   C0 29              CPY #')             ;AT END?...
A17F   F0 07              BEQ SMOVR           ;YES...EXIT TO MOVER
A181   C0 2C              CPY #',             ;MUST HAVE GOTTEN A SEPERATOR
A183   F0 E2              BEQ SLOOP           ;YES...CONTINUE
A185                    ;
A185   4C 08 CF  SDERR    JMP SNERR           ;SYNTAX ERROR
A188                    ;
A188   CA        SMOVR    DEX
A189   BD 01 01  SMOVL    LDA FBUFFR+1,X  ;MOVE TO VIC REGS
A18C   9D 0A 90           STA VICREG+10,X
A18F   CA                 DEX
A190   10 F7              BPL SMOVL
A192   60                 RTS
```

248

```
A193                     ; PDL(X) OR PDL X ; X=O OR 1
A193                     ;
A193   20 9E D7    PDL     JSR GETBYT
A196   EO 02               CPX #2          ;ONLY O OR 1
A198   90 03               BCC *+5         ;OKAY
A19A   4C 48 D2    PDLERR  JMP FCERR       ;ILLEGAL QUANTITY
A19D   BD OB 90            LDA VICREG+8,X
A1AO   AB                  TAY
A1A1   4C A2 D3            JMP SNGFLT      ;MAKE IT A NUMBER
A1A4                     ;
A1A4                     ; JOY (X) XMUST BE A BYTE VALUE
A1A4                     ;
A1A4   20 9E D7    JOY     JSR GETBYT
A1A7   78          REDJOY  SEI             ;CANNOT INTERRUPT
A1A8   A2 7F               LDX #$7F
A1AA   8E 22 91            STX D2DDRB
A1AD   AC 20 91    JOYLP1  LDY D2ORB       ;GET JOY3
A1BO   CC 20 91            CPY D2ORB
A1B3   DO F8               BNE JOYLP1
A1B5                     ;
A1B5   A2 FF               LDX #$FF        ;RESET DDRB
A1B7   8E 22 91            STX D2DDRB
A1BA   A2 F7               LDX #$F7        ;RESTORE STOP KEY CHECK
A1BC   8E 20 91            STX D2ORB
A1BF   58                  CLI             ;RESTORE IRQ'S
A1CO                     ;
A1CO   AD 1F 91    JOYLP2  LDA D1ORA       ;GET JOYO,1,2 & BUTTON
A1C3   CD 1F 91            CMP D1ORA
A1C6   DO F8               BNE JOYLP2
A1C8                     ;
A1C8   48                  PHA
A1C9   29 1C               AND #%00011100  ;MASK OF JOYS
A1CB   4A                  LSR A           ;MOVE DOWN ONE
A1CC   CO 80               CPY #$80        ;CHECK FOR A JOY3
A1CE   90 02               BCC JOYLP3
A1DO   09 10               ORA #%00010000  ;SET, TURN ON
A1D2   A8          JOYLP3  TAY             ;MOVE TO TEMP
A1D3   68                  PLA
A1D4   29 20               AND #%00100000  ;MASK ON BUTTON
A1D6   C9 20               CMP #%00100000  ;CHECK FOR EXISTANCE
A1D8   98                  TYA
A1D9   6A                  ROR A           ;MOVE SO 7=BUTTON 3210=JOYS
A1DA   49 8F               EOR #%10001111  ;FLIP SO POS LOGIC
A1DC   A8                  TAY
A1DD   4C A2 D3            JMP SNGFLT      ;MAKE IT A NUMBER
A1EO                       END
A1EO                       LIB PLOT
A1EO                GRPNT   =$FD
A1EO                TEMP1   =$FB
A1EO                TEMP2   =$FC
A1EO                GRSCRN  =$1000
A1EO                COLLEN  =160
A1EO                     ;
A1EO                     ; HI RES PLOT LOGIC
A1EO                     ;
```

```
LOC     CODE            LINE


A1E0                    SETPLT
A1E0    A2 00               LDX #0              ;MOVE TOP OF MEMORY BELOW HI REG
A1E2    A0 10               LDY #$10
A1E4    18                  CLC
A1E5    20 99 FF            JSR MEMTOP
A1E8    A9 1E               LDA #$1E            ;SCREEN RAM WILL BE AT $1E00
A1EA    85 FE               STA GRPNT+1
A1EC    86 FD               STX GRPNT
A1EE    8A                  TXA                 ;SET UP SCREEN SO ALL CHARACTERS
A1EF    A8                  TAY                 ;START WITH CHARACTER O
A1F0    48          SCRNL   PHA                 ;SAVE "BASE" CHARACTER/POINTER
A1F1    A2 13               LDX #19             ;19 CHARACTERS PER LINE
A1F3    91 FD       SCRNM   STA (GRPNT)Y        ;PUT THE CHARACTER POINTER
A1F5    18                  CLC
A1F6    69 0A               ADC #10
A1F8    C8                  INY
A1F9    D0 02               BNE OK
A1FB    E6 FE               INC GRPNT+1
A1FD                OK
A1FD    CA                  DEX
A1FE    D0 F3               BNE SCRNM
A200    68                  PLA                 ;GET BASE CHAR
A201    18                  CLC
A202    69 01               ADC #1
A204    C9 0B               CMP #11             ;ALL 190 CHARACTERS UP YET?
A206    D0 E8               BNE SCRNL


A208    A9 15               LDA #$15            ;SET VIC FOR 10 ROWS
A20A    8D 03 90            STA VICREG+3
A20D    AD 02 90            LDA VICREG+2        ;GET CURENT COLUMNS
A210    29 80               AND #$80            ;KEEP THIS BIT
A212    09 13               ORA #$13            ;OR IT WITH 19 COLUMS
A214    8D 02 90            STA VICREG+2
A217    A9 F0               LDA #$F0
A219    2D 05 90            AND VICREG+5        ;MAKE SURE KATAKANA IS OFF
A21C    09 0C               ORA #$0C            ;PUT VIC IN HI-RES AT $1000
A21E    8D 05 90            STA VICREG+5        ;SET VIC CHAR ADRS TO $1000
A221                SETCOL
A221    A2 02               LDX #2
A223    A0 00               LDY #0
A225    AD 96 02            LDA COLOR
A228                CLOOP
A228    99 00 96            STA $9600,Y
A22B    99 00 97            STA $9700,Y
A22E    C8                  INY
A22F    D0 F7               BNE CLOOP
A231
A231    A9 10               LDA #$10
A233    85 FE               STA GRPNT+1
A235    A9 00               LDA #0
A237    85 FD               STA GRPNT
A239    A8                  TAY
A23A    A2 0E               LDX #14             ;14 PAGES
```

250

```
LOC    CODE         LINE


A23C                CLRIT
A23C   91 FD            STA (GRPNT)Y
A23E   C8               INY
A23F   DO FB            BNE CLRIT
A241   E6 FE            INC GRPNT+1
A243   CA               DEX
A244   DO F6            BNE CLRIT
A246   60               RTS


A247                PLOT
A247   20 9E D7         JSR GETBYT
A24A   EO 98            CPX #152
A24C   90 02            BCC DTAOK
A24E   A2 97            LDX #151
A250                DTAOK
A250   86 FB            STX TEMP1
A252   20 FD CE         JSR CHKCOM
A255   20 9F D7         JSR GETBYT
A258   EO AO            CPX #160
A25A   90 02            BCC YISOK
A25C   A2 9F            LDX #159
A25E                YISOK
A25E   86 FC            STX TEMP2
A260                                         ;
A260   A5 FB            LDA TEMP1        ;GET X VALUE
A262   4A               LSR A            ;DIVIDE BY 8
A263   4A               LSR A            ;TO GET TABLE INDEX
A264   4A               LSR A
A265   0A               ASL A            ;MAKE IT AN ADDRESS INDEX
A266   AA               TAX
A267   BD 88 A2         LDA GRTBLE,X     ;GET LO BYT OF COLUMN POINTER
A26A   85 FD            STA GRPNT        ;POINT INDIRECT HERE
A26C   BD 89 A2         LDA GRTBLE+1,X
A26F   85 FE            STA GRPNT+1
A271
A271   A5 FB            LDA TEMP1        ;GET THE BIT TO SET
A273   29 07            AND #7
A275   AA               TAX
A276   BD 80 A2         LDA XBITS,X      ;GET BIT FROM TABLE
A279   A4 FC            LDY TEMP2        ;GET ROW INDEX
A27B   11 FD            ORA (GRPNT)Y     ;SET THE BIT
A27D   91 FD            STA (GRPNT)Y     ;DISPLAY IT
A27F   60               RTS


A280   80          XBITS   BYT $80,$40,$20,$10,$08,$04,$02,$01
A281   40
A282   20
A283   10
A284   08
A285   04
A286   02
A287   01
```

LOC	CODE	LINE

A288		COL0=GRSCRN
A288		COL1=COL0+COLLEN
A288		COL2=COL1+COLLEN
A288		COL3=COL2+COLLEN
A288		COL4=COL3+COLLEN
A288		COL5=COL4+COLLEN
A288		COL6=COL5+COLLEN
A288		COL7=COL6+COLLEN
A288		COL8=COL7+COLLEN
A288		COL9=COL8+COLLEN
A288		COL10=COL9+COLLEN
A288		COL11=COL10+COLLEN
A288		COL12=COL11+COLLEN
A288		COL13=COL12+COLLEN
A288		COL14=COL13+COLLEN
A288		COL15=COL14+COLLEN
A288		COL16=COL15+COLLEN
A288		COL17=COL16+COLLEN
A288		COL18=COL17+COLLEN

A288	00 10	CRTBLE .WOR COL0
A28A	A0 10	.WOR COL1
A28C	40 11	.WOR COL2
A28E	E0 11	.WOR COL3
A290	80 12	.WOR COL4
A292	20 13	.WOR COL5
A294	C0 13	.WOR COL6
A296	60 14	.WOR COL7
A298	00 15	.WOR COL8
A29A	A0 15	.WOR COL9
A29C	40 16	.WOR COL10
A29E	E0 16	.WOR COL11
A2A0	80 17	.WOR COL12
A2A2	20 18	.WOR COL13
A2A4	C0 18	.WOR COL14
A2A6	60 19	.WOR COL15
A2A8	00 1A	.WOR COL16
A2AA	A0 1A	.WOR COL17
A2AC	40 1B	.WOR COL18

252

6502 Instruction Set — Hex and Timing

MNEMONIC		IMPLIED			ACCUM.			ABSOLUTE			ZERO PAGE			IMMEDIATE			ABS. X		
		OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#
A D C	(I)							6D	4	3	65	3	2	69	2	2	7D	4	3
A N D	(I)							2D	4	3	25	3	2	29	2	2	3D	4	3
A S L					0A	2	1	0E	6	3	06	5	2				1E	7	3
B C C	(2)																		
B C S	(2)																		
B E Q	(2)																		
B I T								2C	4	3	24	3	2						
B M I	(2)																		
B N E	(2)																		
B P L	(2)																		
B R K		00	7	1															
B V C	(2)																		
B V S	(2)																		
C L C		18	2	1															
C L D		D8	2	1															
C L I		58	2	1															
C L V		B8	2	1															
C M P								CD	4	3	C5	3	2	C9	2	2	DD	4	3
C P X								EC	4	3	E4	3	2	E0	2	2			
C P Y								CC	4	3	C4	3	2	C0	2	2			
D E C								CE	6	3	C6	5	2				DE	7	3
D E X		CA	2	1															
D E Y		88	2	1															
E O R	(I)							4D	4	3	45	3	2	49	2	2	5D	4	3
I N C								EE	6	3	E6	5	2				FE	7	3
I N X		E8	2	1															
I N Y		C8	2	1															
J M P								4C	3	3									
J S R								20	6	3									
L D A	(I)							AD	4	3	A5	3	2	A9	2	2	BD	4	3
L D X	(I)							AE	4	3	A6	3	2	A2	2	2			
L D Y	(I)							AC	4	3	A4	3	2	A0	2	2	BC	4	3
L S R					4A	2	1	4E	6	3	46	5	2				5E	7	3
N O P		EA	2	1															
O R A								0D	4	3	05	3	2	09	2	2	1D	4	3
P H A		48	3	2															
P H P		08	3	1															
P L A		68	4	1															
P L P		28	4	1															
R O L					2A	2	1	2E	6	3	26	5	2				3E	7	3
R O R					6A	2	1	6E	6	3	66	5	2				7E	7	3
R T I		40	6	1															
R T S		60	6	1															
S B C	(I)							ED	4	3	E5	3	2	E9	2	2	FD	4	3
S E C		38	2	1															
S E D		F8	2	1															
S E I		78	2	1															
S T A								8D	4	3	85		2				9D	5	3
S T X								8E	4	3	86		2						
S T Y								8C	4	3	84		2						
T A X		AA	2	1															
T A Y		A8	2	1															
T S X		BA	2	1															
T X A		8A	2	1															
T X S		9A	2	1															
T Y A		98	2	1															

253

ABS. Y			(IND. X)			(IND)Y			Z.PAGE,X			RELATIVE			INDIRECT			Z.PAGE,Y			PROCESSOR STATUS CODES	
OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	N V	BDIZC
79	4	3	6I	6	2	7I	5	2	75	4	2										• •	• •
39	4	3	2I	6	2	3I	5	2	35	4	2										•	•
									I6	6											•	• •
												90	2	2								
												BO	2	2								
												FO	2	2								•
												30	2	2								
												DO	2	2								
												IO	2	2								
												50	2	2								
												70	2	2								
D9	4	3	CI	6	2	DI	5	2	D5	4	2										•	• •
																					•	• •
																					•	• •
									D6	6	2										•	•
59	4	3	4I	6	2	5I	5	2	55	4	2										•	•
									F6	6	2										•	•
															6C	5	3				•	•
B9	4	3	AI	6	2	B1	5	2	B5	4	2										•	•
BE	4	3							B4	4	2							B6	4	2	•	•
									56	6	2										•	• •
I9	4	3	OI	6	2	II	5	2	I5	4	2										•	•
																					• •	• • • • • •
									36	6	2										•	• •
									76	6	2										•	• • • • • •
F9	4	3	EI	6	2	FI	5	2	F5	4	2										• •	• •
99	5	3	8I	6	2	9I	6	2	95	4	2							96	4	2		
									94	4	2											
																					•	•
																					•	•
																					•	•
																					•	•

254

HEXADECIMAL CONVERSION TABLE

HEX	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
8	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
9	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
A	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
B	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
C	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
D	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
E	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
F	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255

	5		4		3		2		1		0
HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC
0	0	0	0	0	0	0	0	0	0	0	0
1	1,048,576	1	65,536	1	4,096	1	256	1	16	1	1
2	2,097,152	2	131,072	2	8,192	2	512	2	32	2	2
3	3,145,728	3	196,608	3	12,288	3	768	3	48	3	3
4	4,194,304	4	262,144	4	16,384	4	1,024	4	64	4	4
5	5,242,880	5	327,680	5	20,480	5	1,280	5	80	5	5
6	6,291,456	6	393,216	6	24,576	6	1,536	6	96	6	6
7	7,340,032	7	458,752	7	28,672	7	1,792	7	112	7	7
8	8,388,608	8	524,288	8	32,768	8	2,048	8	128	8	8
9	9,437,184	9	589,824	9	36,864	9	2,304	9	144	9	9
A	10,485,760	A	655,360	A	40,960	A	2,560	A	160	A	10
B	11,534,336	B	720,896	B	45,056	B	2,816	B	176	B	11
C	12,582,912	C	786,432	C	49,152	C	3,072	C	192	C	12
D	13,631,488	D	851,968	D	53,248	D	3,328	D	208	D	13
E	14,680,064	E	917,504	E	57,344	E	3,584	E	224	E	14
F	15,728,640	F	983,040	F	61,440	F	3,840	F	240	F	15

Circuit Diagrams
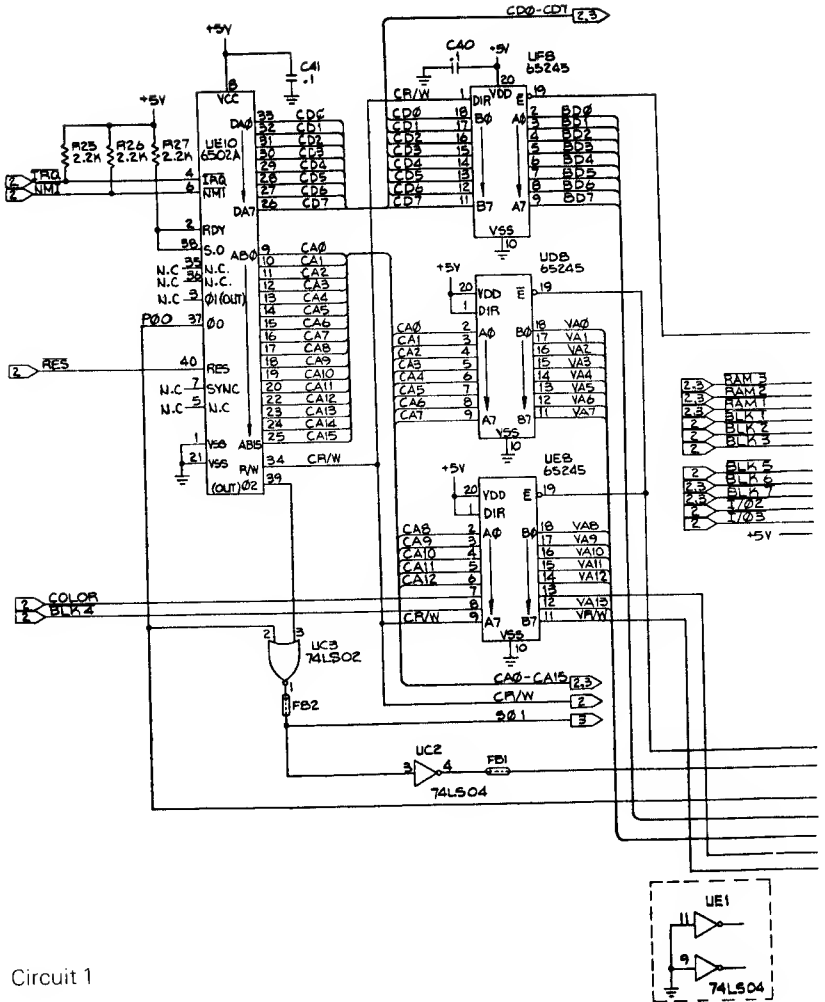
Circuit 1
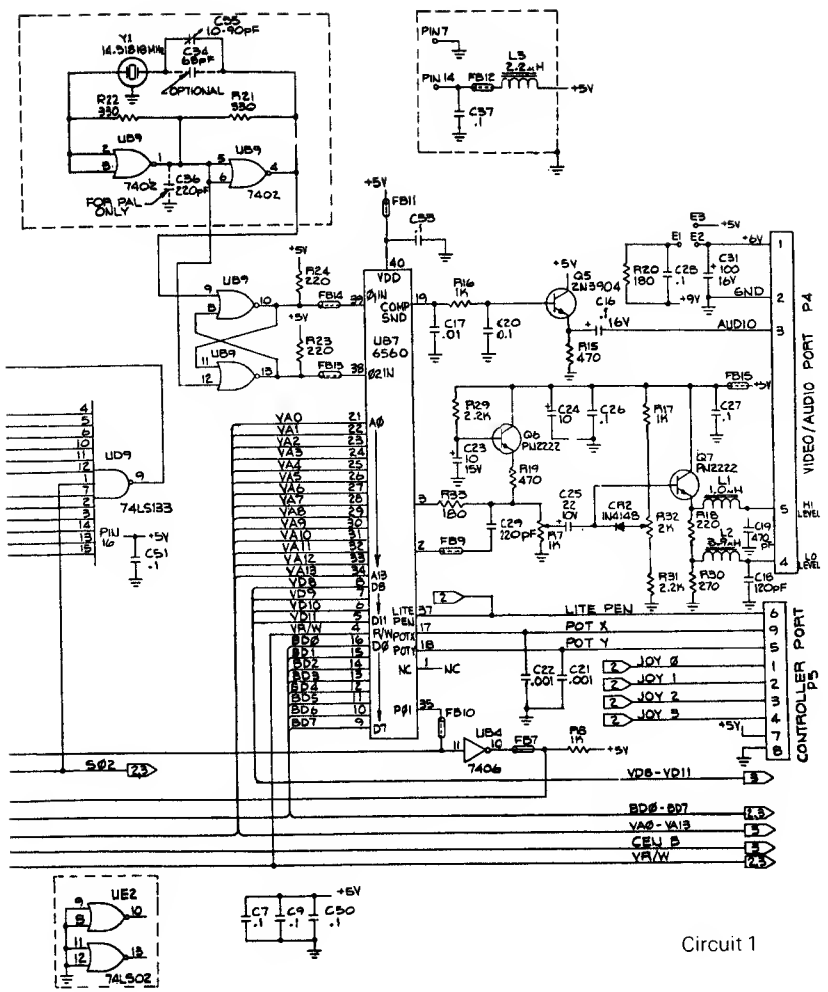
Circuit 1

Circuit 2

Circuit 2

Circuit 3

261

Circuit 3

+5V

C11 C49 C14

+5V C42   +5V C43

18 VCC VSS 9   18 VCC VSS 9   18 VCC VSS 9   18 VCC VSS 9   18 VCC VSS 9
     CS 8   CS 8   CS 8   CS 8   CS 8

AØ   AØ   AØ   AØ   AØ

24   24

UD4   UE5   UD5   UE6   UD6
2114   2114   2114   2114   2114

A9   A9   A9   A9   A9
R/W   R/W   R/W   R/W   R/W
DØ D1 D2 D3   DØ D1 D2 D3   DØ D1 D2 D3   DØ D1 D2 D3   DØ D1 D2 D3
11 12 13 14   11 12 13 14   11 12 13 14   11 12 13 14   11 12 13 14

CAØ 8   AØ   8   AØ
CA1 7   7
CA2 6   6
CA3 5   5
CA4 4   4
CA5 3   3
CA6 2   2
CA7 1   1
CA8 23   23
CA9 22   22
CA10 19   19
CA11 18   18
CA12 21 A12   21 A12

UE11   UE12
2364   2364

CDØ 9   DØ   9   DØ
CD1 10   10
CD2 11   11
CD3 13   13
CD4 14   14
CD5 15   15
CD6 16   16
CD7 17 D7   17 D7
     20 CS   VSS   20 CS   VSS
          12   12

+5V C38

24   VCC

VAØ 8   AØ
VA1 7
VA2 6
VA3 5
VA4 4
VA5 3
VA6 2
VA7 1
VA8 23
VA9 22
VA10 19
VA11 18 A11
VA12 21 CS2
VA13 20 CS1

UD7
2332

BDØ 9   DØ
BD1 10
BD2 11
BD3 13
BD4 14
BD5 15
BD6 16
BD7 17 D7

VSS
12

APPENDIX 6
COMMANDS USED BY VIC MACHINE CODE MONITOR
All commands are displayed in BOLD TYPE

simple assembler
.a 2000 a9 12   lda No. $12
.A 2002 9D 00 80 STA $8000,X
.A 2005 **DEX** GARBAGE

In the above example the user started assembly at 2000 hex. The first instruction was load a register with immediate 12 hex. In the second line the user did not need to type the A and address. The simple assembler retyped the last entered line and prompts with the next address. To exit the assembler type a return after the address prompt. Syntax is the same as the disassembler output. A ':' can be use to terminate a line.

disassembler
.d 2000

., 2000 A9 12   LDA No. $12
., 2002 9D 00 80 STA $8000,X
., 2005 AA   TAX

disassembles to the end of memory starting at 1000 hex. The three bytes following the address may be modified. Use the CRSR keys to move to and modify the bytes. Hit return and the bytes in memory will be changed. Monitor will then disassemble that line again.

.d 2000 3000

disassembles from 2000 to 3000.

fill memory
.f 1000 1100 ff

fills the memory from 1000 hex to 1100 hex with the byte ff hex.

go run
.g

go to the address in the pc register display and begin run code. All the registers will be replaced with the displayed values.

.g 1000

go to address 1000 hex and begin running code.

hunt memory
.h c000 d000 'read

hunt through memory from c000 hex to d000 hex for the ASCII string **read** and print the address where it is found. A maximum of 32 characters may be used.

.h c000 d000 20 d2 ff

hunt memory from c000 hex to d000 hex for the sequence of bytes 20 d2 ff and print the address. A maximum of 32 bytes may be used. Hunt can be stopped with the stop key.

integerate memory
.i f000

.' f000 54 4f 4f 20 4d 41 4e 59 **too many**
.' F008 20 46 49 4C 45 D3 46 49 **FILES FI**

displays hex and ascii until the end of memory.

.i f000 f080

displays hex and ascii from f000 hex to f080 hex.

load from tape
.l

load any program from cassette No. 1.

.l "ram test"

load from cassette No. 1 the program NAMED **RAM TEST**

.l "ram test" ,02

load from cassette No. 2 the programme NAMED **RAM TEST**

beware load with a file name breaks the irq saved by the monitor. Do not use go command after load or save. Exit to basic and re-enter monitor.

memory display
.m 0000 0080

.: 0000 00 01 02 03 04 05 06 07
.: 0008 08 09 0A 0B 0C 0D 0E 0F

display memory from 0000 hex to 0080 hex. The bytes following the address may be modified by editing and then typing a return.

new locater
.n 7000 77ff 1000 0400 8000
.n 7000 77ff 1000 0400 8000 W

relocates machine code from 7000 hex to 77ff hex to a new location at 1000 hexf. New locater fixes all 3 byte instructions in the range 0400 hex to 8000 hex. The 'W' option will relocate word tables only. New locater will not move instructions of 00. Transfer the tables first then zero tables in the form copy. New locater stops and disassembles on a bad op code.

register display
.r

pc sr ac xr yr sp

., 0000 01 02 03 04 05

displays the register values saved when monitor was entered. The values may be changed with the edit followed by a return.

use this instruction to set up the pc value before single stepping with.

save to tape
.s "program name",01 ,0800 ,0c80

save to cassette No. 1 memory from 0800 hex up to but not including 0c80 hex and name it program name.

beware save with a file name breaks the irq saved by the monitor. Do not use go command after load or save exit to basic and re-enter monitor.

walk code
.w

single step starting at address in register pc.

.w 1000

single step starting at address 1000 hex. Walk will cause a single step to execute and will disassemble the next instruction.

control speed with choice of key:

K for single step;
RVS for slow step:
SPACE for fast stepping

exit to basic
.X

return to basic ready mode. The stack value saved when entered will be restored. Care should be taken that this value is the same as when the monitor was entered. A clr in basic will fix any stack problems.

.

265

INDEX